National Library of Canada
Acquisitions and Bibliographic Services Branch
395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale du Canada
Direction des acquisitions et des services bibliographiques
395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Canada

UNIVERSITY OF ALBERTA

CLOCKED RATIOED LOGIC
FOR
FINITE STATE MACHINE IMPLEMENTATION

©

BY

MAGGIE, YEE MAN PONG

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of Master of Science.

DEPARTMENT OF ELECTRICAL ENGINEERING

Edmonton, Alberta

FALL, 1993

National Library of Canada
Acquisitions and Bibliographic Services Branch
395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale du Canada
Direction des acquisitions et des services bibliographiques
395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-88380-4

Canada

UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: Maggie, Yee Man Pong

TITLE OF THESIS: Clocked Ratioed Logic for FSM Implementation

DEGREE: Master of Science

YEAR THIS DEGREE GRANTED: 1993

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as hereinbefore provided neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.

Maggie Pong

11643, 84 Street
Edmonton, Alberta
Canada

Date: 8-10-93

UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Clocked Ratioed Logic for Finite State Machine Implementation** submitted by **Maggie, Yee Man Pong** in partial fulfillment of the requirements for the degree of **Master of Science**.

Supervisor: Dr. J. T. Mowchenko

Dr. J. Schaeffer

Dr. K. A. Stromsmoe

Date: Oct 8, 1993

Abstract

Digital systems have become an irreplaceable part of modern societies. As the functions performed by digital systems become more sophisticated, the control units of these systems become so complex that they dominate the performance and the cost of the systems. Therefore, this thesis addresses the problem of finite state machine (FSM) implementation, and try to find a way to optimize the size and the speed of the FSMs.

Currently, FSMs have been implemented using PLAs, standard cells, or as microprogrammable FSMs. However, all of the FSM implementation methods have their limitations. The new FSM implementation approach that is presented in this thesis involves merging the combinational logic and the memory elements of the FSMs together. This is made possible by using a new type of logic called CLOcked RAtioed logic (CLORA). Moreover, this new approach uses one-hot state assignment so that it can be used to implement FSMs that have concurrent operation sequences. Based on this new approach, a FSM generator is developed under the Cadence (a computer aided design system) design environment.

Experiments have been performed to compare the FSM size obtained using the PLA implementation and the new method. It is found that the area of the un-encoded FSM implementation using our new method is on average about 1.2 times the area given by the fully-encoded PLA implementation. On the other hand, with fully-encoded FSMs, the results given by our new method is comparable to that of the PLA implementation. Also, through hspice simulation, we found that FSMs implemented using the new method can run at 80MHz.

Acknowledgement

I would like to thank my supervisor, Dr. J. Mowchenko, for his guidance throughout the research. Also, I would like to acknowledge the financial support provided by the Alberta Microelectronics Corporation and the NSERC. Moreover, I would like to thank the Canadian Microelectronics Corporation for providing the Cadence software used during the project.

Table of Contents

CHAPTER 1

Introduction **1**

1.1 New FSM Implementation Method 3

1.2 Thesis Overview 4

CHAPTER 2

Background **5**

2.1 Logic Minimization of FSM 5

2.2 Multi-level FSM Logic Minimization 6

2.2.1 MIS 6

2.2.2 MUSTANG 7

2.3 Two-Level FSM Logic Minimization 8

2.3.1 ESPRESSO II 8

2.3.2 NOVA 10

2.4 Evaluation of State Assignment 11

2.5 Layout Structures for FSMs 12

2.6 Regular Arrays 12

2.6.1 Programmable Logic Array (PLA) 13

2.6.2 Gate Matrix 14

2.6.3 Weinberger Array 15

2.6.4 Layout Structure Realization 17

2.7 Standard Cells 19

2.8 Counter-Based FSMs 20

2.9 Microprogrammable FSM 22

2.10 Evaluation of Current Layout Structures 23

2.11 Summary 23

CHAPTER 3

New FSM Implementation **25**

3.1 Problem Definition 25

3.2 Clocked Ratioed Logic Gate 28

3.2.1 Operation of CLORA Logic 29
3.2.2 Designing the CLORA Logic 29
3.2.3 Timing Characteristic of a CLORA Inverter 33
3.2.4 Pros and Cons of CLORA Logic 33
3.3 FSM Realization 34
3.3.1 Un-encoded FSM realization 34
3.4 The Operation of a CLORA FSM 36
3.5 Reset Mechanism 37
3.6 Pros and Cons of the CLORA FSM 38
3.7 Layout Structure of a CLORA FSM 39
3.7.1 Gate Schematic 41
3.7.2 Clock Buffers 43
3.7.3 Schematic of the CLORA Array 45
3.7.4 Physical Layout of the CLORA Array 47
3.7.5 Pros and Cons of the CLORA Array 48
3.8 Fully-Encoded FSM realization 49
3.9 Summary 50

CHAPTER 4

Implementation 51
4.1 Cadence 51
4.2 Overview of Programmable Logic Array Layout Generator ... 54
4.3 Anchor's Output 55
4.4 Folded PLA MM Generator 57
4.4.1 The Init Stage 58
4.4.2 OCTRead 59
4.4.3 The Layout Stage 60
4.4.4 Folded PLA template 60
4.4.5 Master Cells 62
4.5 Overview of the CLORA FSM Layout Generator 62
4.6 clora 64
4.6.1 Read KISS 64
4.6.2 Read PLA 67
4.6.3 Gate Ordering 68
4.6.4 Layout Gen 70

4.7 CLORA MM Generator 72
4.7.1 The Init Stage 72
4.7.2 ReadFile 74
4.7.3 The Layout Stage 74
4.7.4 CLORA Array template 74
4.7.5 Master Cells 76
4.8 Summary 76

CHAPTER 5

Results 78
5.1 clora's Results 78
5.1.1 Un-encoded FSM Implementation 79
5.1.2 Fully-encoded FSM Implementation 81
5.2 Results of Physical Layouts 81
5.3 Speed Performance 87
5.4 Summary 87

CHAPTER 6

Conclusion 88
References 91

List of Tables

TABLE 2.1 Standard Cells Layout vs. PLA....20

TABLE 3.1 Transfer Characteristic for a CLORA Gate with Different Aspect Ratios. 32

TABLE 5.1 Information of FSMs. 78

TABLE 5.2 clora's Results of the Un-encoded FSM Implementation With HR Plane.79

TABLE 5.3 clora's Results of the Un-encoded FSM Implementation Without HR Plane....79

TABLE 5.4 clora's Results of the Fully-encoded FSM Implementation. 81

TABLE 5.5 Actual Area of Various Implementations in DSM2....82

List of Figures

FIGURE 2.1 Basic Structure of a PLA. 13
FIGURE 2.2 Basic Gate Matrix Layout 15
FIGURE 2.3 Basic Structure of a Weinberger Array. 16
FIGURE 2.4 Basic Domino Logic Structure 17
FIGURE 2.5 Basic NORA Logic Structure 18
FIGURE 2.6 Basic Standard Cells Layout 20
FIGURE 2.7 Schematic of a Counter-Based FSM 21
FIGURE 2.8 Microprogrammable FSM. 22
FIGURE 3.1 Cascaded Dynamic Logic 26
FIGURE 3.2 Clocked Inverter. 27
FIGURE 3.3 A CLORA Inverter. 29
FIGURE 3.4 Circuit Used in Determining Aspect Ratio. 30
FIGURE 3.5 Input Waveforms Used in Simulation 30
FIGURE 3.6 Output Waveforms of a CLORA Inverter. 31
FIGURE 3.7 CLORA inverter with CL. 33
FIGURE 3.8 Sample FSM and its NOR gate realization 35
FIGURE 3.9 Edge and State Gates of a CLORA FSM. 36
FIGURE 3.10 Timing Diagram for F1, F2 and RESET. 37
FIGURE 3.11 State Gate for Starting State 37
FIGURE 3.12 A CLORA NOR Gate 39
FIGURE 3.13 A CLORA FSM Structure. 40
FIGURE 3.14 Schematic of a Edge Gate and a Static CMOS inverter 41
FIGURE 3.15 Schematic of a Horizontal State Gate. 42
FIGURE 3.16 Schematic of a T-Shaped State Gate 42
FIGURE 3.17 Schematic of a Vertical Output Gate. 43

FIGURE 3.18 Schematic of the Clock Buffer Planes. 44
FIGURE 3.19 Structure of a CLORA Array....46
FIGURE 3.20 A Transistor in the CLORA Array....47
FIGURE 3.21 A Clocked Transistor in the CLORA Array. 48
FIGURE 4.1 A Sample Template Used by the Structure Complier....52
FIGURE 4.2 A Sample Personality File Used by the Structure Complier. 53
FIGURE 4.3 a) A Sample .andor File;
b) A Sample .input File;
c) A Sample .output file....56
FIGURE 4.4 Folded PLA Template....60
FIGURE 4.5 Data Flow in the CLORA FSM Layout Generator. 63
FIGURE 4.6 Sample FSM description in KISS Format. 64
FIGURE 4.7 The Combinational Logic of an FSM in PLA Format....68
FIGURE 4.8 Sample Genie Input File. 70
FIGURE 4.9 a) Sample .array File;
b) Sample .io File Generated by clora. 72
FIGURE 4.10 CLORA Array Template....74
FIGURE 4.11 Physical Layout of the With-HR Implementation of the Un-encoded Opus. 84
FIGURE 4.12 Physical Layout of the Without-HR Implementation of the Un-encoded Opus. 85
FIGURE 4.13 Physical Layout of the Fully-encoded Opus....86

CHAPTER 1

Introduction

Digital systems consist of two parts: the data path and the control path. The data path consists of functional units which operate on the data and perform the work for the system. These functional units usually are implemented as combinational circuits. This means that their output values depend only on the present values of the inputs. The control path receives status signals as inputs and produces control signals to control the operation of the data path. The control path is usually implemented as a synchronous sequential circuit and can be modelled as a finite state machine (FSM). For an FSM, the output values depend on both the present and past input values.

Many computer-aided design packages have been developed for designing digital systems. Most of these packages tend to implement the controller of the digital system as a programmable logic array (PLA) because of simplicity of design and implementation. However, because of its low device density, and because its size depends heavily on state assignment, the PLA may not be the best way to implement controllers. In this project, we will try to come up with a new layout method that requires no state assignment and, hopefully, can lay out controllers as effectively as PLA's, if not better.

Presently, digital systems can be found anywhere from personal computers to banking machines. They have become an irreplaceable part of modern societies. As the strong public demands for more sophisticated digital systems grow, the tasks performed by the digital systems become more complex, thus increasing the size of the digital systems. However, no matter how many functions a digital system can perform, if the size of the digital system is too great, it will be uneconomical or impractical. For example, no one would want to carry around a cellular phone as big as a telephone book. Thus, because of the importance of digital systems, a considerable amount of research has been done on finding the best way to implement them such that their size is minimized. One line of research concentrates on developing better CMOS processing technologies such that the

line width of the layout of the digital system is reduced. Thus, using a more advanced CMOS technology, a digital system can be laid out using less area. Another line of research concentrates on finding better methods for implementing or for laying out the digital systems. This research includes logic minimization, floorplanning, logic structures, and high-level synthesis.

As the functions performed by digital systems become more sophisticated, the control units of these systems become so complex that they dominate the performance and the cost of the system. Therefore, it is very important to find a way to optimize the size and the speed of the control unit of a digital system. Current work on optimizing the control path (FSM) has been concerned with either optimizing the state assignment of the FSM or developing better FSM synthesis systems. The former involves the encoding of the FSM states such that the resulting FSM logic is minimized. The latter involves transforming a high-level specification of an FSM into the physical layout of the FSM based on a specific control structure. Both FSM optimization techniques have the intention of implementing the FSM using standard cells, PLA, or as a microprogrammable FSM. However, not much work has been done on finding an efficient layout method for FSMs. Are PLAs, standard cells, and microprogrammable FSMs the best implementation methods? To implement an FSM using a PLA, one needs to add extra flip-flops to the PLA to make it a sequential circuit. Also, the device packing density of PLAs is fairly low. For a standard cell implementation of an FSM, the standard cells use static CMOS logic which takes up more area than other logic structures. Moreover, for either of these implementations to give a reasonable sized FSM, the states of the FSM must be encoded using state assignment so that the number of state variables is reduced. However, once the states of the FSM are encoded, the testability of the FSM decreases. In the case of a microprogrammable FSM, although it does not require state assignment, it requires a larger area to implement. Also, since each state transition involves executing an instruction in the memory, the speed of the FSM is impeded. Thus, these three commonly used FSM implementations may not be the best ways of implementing an FSM after all.

1.1 New FSM Implementation Method

To find a better way to implement FSMs, we have decided to use a regular array structure so that the FSM generation process can be easily automated. Also, we have decided to use one-hot state assignment (i.e. one state variable per state) in our new FSM implementation method. According to [1], testing an un-encoded FSM is simpler than that of an encoded FSM. This is because testing an encoded FSM requires scan path while testing an un-encoded FSM only requires test patterns to the sensitized paths in the FSM and these can be easily generated by inspecting the state table. For further details on testing encoded and un-encoded FSM, readers can refer to [1]. Since testability is an increasing concern in digital system design, it is very important to choose an approach that does not impede the testability of the FSMs.

Un-encoded FSMs also allow us to implement FSMs which have concurrent operation sequences (i.e. FSMs that can be in more than one state simultaneously) [2]. Moreover, since many of the signals will only be used in a local part of the FSM, the relationship between the signals in an un-encoded FSM will not be as complex as that in the encoded FSM. Therefore, the signal nets in the FSM are shorter. This holds a big advantage in a regular array implementation because the size of the regular array heavily depends on the length of the signals in the array. If the lengths of most of the signals in the array are short, they can be arranged more compactly, thus reducing the size of the array.

Another important feature that our FSM implementation method has is that it merges the combinational and sequential logic of an FSM together. Thus far, in many of the different FSM layout structures, FSMs are implemented as a combinational logic block with a sequential logic block attached to it. The problem with this approach is that area is wasted in interconnecting the combinational and the sequential logic blocks together. Also, in most of the FSMs, the sequential logic blocks are made up of standard cells. Thus, the sequential logic block may take up a significant amount of area in the FSM. To merge the combinational and the sequential logic together, we incorporated the memory elements of an FSM within the regular array which evaluates the combinational logic of the FSM. To

achieve this, we use the CLOcked RAtioed logic (CLORA) to implement the regular array. The CLORA logic was developed as part of the research project.

1.2 Thesis Overview

This thesis is devoted to describing the new layout method and comparing the new method with the PLA FSM implementation method. In Chapter 2, some of the current FSM implementation methods are reviewed. Then, Chapter 3 will give a detailed description of the new FSM implementation method that we have developed. Although this chapter solely deals with non-concurrent FSMs, similar concept applies also to concurrent FSMs. Following that, Chapter 4 will give a detailed description on how the new FSM implementation method is automated under the design environment of Cadence which is a computer aided design system. Chapter 5 will give the result obtained by implementing some of the sample FSMs provided by Microelectronics Center of North Carolina (MCNC) using the new FSM implementation method. In addition, a comparison between the new CLORA array FSM implementation and the PLA FSM implementation in terms of area is done in order to see how this new method performs.

CHAPTER 2

Background

Since finite state machines (FSM) are at the heart of every digital system, a considerable amount of research has been done on finding a method to efficiently implement FSMs. This chapter reviews some of these methods and discusses their relative advantages and disadvantages.

The conventional way of implementing an FSM usually requires two stages: the logic optimization stage and the layout stage. Thus, this chapter is divided into two main sections. The first section deals with ways to minimize the logic of an FSM. In this section, two state assignment algorithms and two logic minimization algorithms are reviewed. The second section deals with ways to lay out an FSM. In this section, various layout structures such as regular arrays and standard cells are described.

2.1 Logic Minimization of FSM

In order to reduce the final size of an FSM, the logic which it implements must be minimized before the layout stage. Logic minimization typically requires two steps. The first step is state assignment, which is the process of assigning a unique binary code to each state of the FSM. To date, this has been one of the most active research areas related to FSM optimization. Since there are numerous ways to lay out an FSM, state assignment algorithms have been tailored to different layout structures. The second step in FSM logic minimization is logic optimization. The logic optimizer also depends on the intended layout structure of FSM. In the following sections, two FSM state assignment systems tailored for two different layout structures are presented along with their intended logic optimizers.

2.2 Multi-level FSM Logic Minimization

FSMs can be implemented using either multi-level logic (i.e. standard cells) or two-level logic (i.e. PLAs). Multi-level FSM logic minimization is the process of simplifying the logic of a set of multi-level logic functions. One of the most commonly used state assignment systems for multi-level FSM implementation is MUSTANG [3]. The following section will give a review of MUSTANG and its intended multi-level logic optimizer, MIS [4].

2.2.1 MIS

MIS is a multi-level logic optimizer which uses a global optimization methodology [4]. It optimizes a set of logic functions by factoring them into an optimal multi-level form independent of the form of the input description. In MIS, a directed acyclic graph called the boolean network, is used to represents a set of multi-level logic functions. Each node i in the boolean network has an associated variable y_i and a representation f_i of a logic function which indicates whether that function is in a sum-of-products or a factored form. An arc connecting node i to node j indicates that y_i appears in f_j.

Three main functions can be performed by MIS. The first one is global area optimization. Global area optimization is independent of the design style or technology. Its goal is to minimize the complexity of the logic equations so that the area needed to implement them is minimized. The complexity of the multi-level logic functions is defined as the sum of the number of literals in the minimum factored form of the function associated with each node in the Boolean network. Thus, to minimize the area of a set of multi-level logic functions, MIS minimizes the number of literals in the factored form of the function associated with each node. The first step in global area optimization is extraction. In this step, MIS identifies common factors present in the multi-level logic equations. The second step is resubstitution. In this step, MIS identifies the common factors that may be missed in the extraction step. It checks to see whether an existing function is a factor of other functions. The last step is phase assignment. In this step, MIS chooses the phase of each

intermediate function (i.e. its true or complementary value) to implement so that the number of inverters needed to implement the logic equations is minimized.

The second function that can be performed by MIS is local optimization. Local optimization is performed on a single node in the boolean network, or on the neighborhood of the node. It decomposes large gates into smaller ones, derives better implementations of the smaller gates using factorization, and locally simplifies the resulting gates.

The third function is timing pre-optimization. This function is performed on networks which have already had their area minimized. According to the timing specification of the network, MIS reduces the delay through the network with minimum area increase.

2.2.2 MUSTANG

MUSTANG is a state assignment system developed for multi-level logic implementation (i.e. standard cells, gate matrix or Weinberger array) of FSMs [3]. The state assignment algorithm is designed with MIS in mind. That is, MUSTANG will find a state assignment such that the area of the FSM after running it through MIS is minimized. The goal of MUSTANG is to find an encoding in the two-level representation of the FSM logic functions that maximizes the number of common cubes among the combinational logic expressions of the FSM. Then, MIS can map the two-level description into a multi-level implementation by factoring and decomposing the logic functions.

To achieve its goal, MUSTANG first constructs a complete graph G_M (V, E_M, $W(E_M)$). In this graph, V is the set of nodes, where each node corresponds to a state in an FSM. E_M is a complete set of edges and $W(E_M)$ is the gain that can be achieved by coding the states joined by E_M as close to each other as possible. The states are then encoded by using this graph to evaluate the cost of the state assignment. Two different algorithms are used to generate $W(E_M)$. The first one is called the fan-out-oriented algorithm. This algorithm gives high edge weights to edges joining present states which assert similar outputs and produce similar sets of next states. Assigning high weights to these edges maximizes the

size of common cubes in the output and next state signals. Since this algorithm ignores the input field, it only works well for FSMs with a large number of outputs and small number of inputs. The second algorithm is the fan-in-oriented algorithm. This algorithm gives high weights to edges joining next states which are produced by similar inputs and similar sets of present states. This algorithm maximizes the number of common cubes in the next state signals.

Once the complete graph is generated, it is used to guide the state encoding process. In MUSTANG, this state encoding process is solved as a combinational optimization problem where G_M has to be embedded in the boolean hypercube so that the adjacency relations identified by G_M are satisfied in an optimal way.

Experimental results have shown that the literal count obtained by MUSTANG is on average 30 percent better than that obtained by random state assignment and 20 percent better than that obtained by KISS [5], a state assignment system for PLA-based FSMs. However, since MUSTANG models only the common cube extraction process in MIS, the resulting state encoding does not take full advantage of MIS.

2.3 Two-Level FSM Logic Minimization

Two-level FSM logic minimization is usually applied to FSMs implemented as Programmable Logic Arrays (PLAs). One of the most commonly used state assignment system for two-level FSM implementation is NOVA [6]. The following section will give a review of NOVA and its intended two-level logic optimizer, ESPRESSO II [7].

2.3.1 ESPRESSO II

ESPRESSO II is a logic minimizer for two-level logic [7]. To help readers understand the algorithms presented in this section, some basic definitions are provided.

- incompletely specified logic function:

  a logic function whose outputs can be either 0, 1, or don't care.

  completely specified logic function:

a logic function whose outputs can be either 0, or 1.

- on-set, off-set, or don't care set of a logic function:

  a set of input values such that the output of the logic function is 1, 0 or don't care, respectively.

- cube:

  a conjunction of a logic functions' input literals.

- cover of a logic function:

  a union of the on-set and part of the don't-care-set of a logic function.

ESPRESSO II receives the cube cover of the on-set, ***F***, and the cube cover of the don't-care set, ***D***, of an incompletely specified Boolean function. It returns a minimized cover of the function as its output. The objectives of ESPRESSO II are to minimize the number of product terms in the cover, the number of literals in the input parts of the cover, and the number of literals in the output parts. The operation sequence of ESPRESSO II involves several procedures. First, ESPRESSO II unravels any output sharing that may be present in the incoming data so as to achieve a more reliable and input independent result. This means that if a cube is a member of the cube cover of the on-sets of more than one output, an identical cube is created for each of these outputs instead of having all these outputs sharing one cube. Next, a procedure called COMPLEMENT is used to compute the cube cover of the off-set, ***R***, based on the given ***F*** and ***D***. The result of this procedure can be used to quickly determine whether a given cube is an implicant. This aids another procedure, EXPAND, in selecting the prime implicants. After COMPLEMENT, the main minimization loop starts. First, EXPAND is used to replace the cubes of ***F*** with prime implicants. Then, its result, except for the first pass, is compared with that of the previous execution of EXPAND. If no improvement is detected, ESPRESSO II exits the minimization loop. In the first pass, a procedure called ESSENTIAL_PRIMES will be called after EXPAND to locate all prime implicants which must appear in every cover of ***F*** (i.e. essential primes), remove them from ***F***, and add them to ***D***. Thus, the essential primes will not be processed during minimization. After that, the quality of the result is compared

with that of the initial one. Again, if no improvement is detected, ESPRESSO II exits the minimization loop. The next procedure to execute in the loop is IRREDUNDANT_COVER. It is used to sort the cover F into relatively essential, partially redundant and totally redundant subcovers. The totally redundant cubes are discarded, and a minimal subset of the partially redundant subcover is retained. After this, the result is compared with that of the last execution of IRREDUNDANT_COVER. If no improvement is detected, ESPRESSO II exits the minimization loop. Following IRREDUNDANT_COVER, procedure REDUCE is used to improve the local minimum obtained by IRREDUNDANT_COVER. It sequentially reduces each implicant to a minimum essential implicant. The reasons of doing this are that a smaller cube can generally be expanded in more directions than a larger cube, and that smaller cubes have more chance of being covered by the expansions of other cubes. Thus, REDUCE allows ESPRESSO II to move away from a locally minimal solution towards a yet smaller cover. After executing REDUCE, the result is compared with that of the previous execution of REDUCE. If no improvement is detected, ESPRESSO II exits the minimization loop. After REDUCE, ESPRESSO II branches back to the beginning of the loop and starts another cycle by executing EXPAND.

Once ESPRESSO II exits the main minimization loop, a procedure called LAST_GASP is executed. It searches systematically in an order independent fashion for additional primes which are beneficial to the cover. If it finds these primes, ESPRESSO II re-enters the main minimization loop starting with the REDUCE procedure. Otherwise, ESPRESSO II executes the last procedure call, MAKE_SPARSE. This procedure includes the essential primes from D back into the cover F. It also makes the PLA structure as sparse as possible so that the PLA can be folded more efficiently.

2.3.2 NOVA

NOVA is a state assignment system designed for two-level logic implementation of FSMs (i.e. PLA) using multiple-valued logic [6]. In a multiple-valued representation, the states of an FSM are represented as the set of possible values for a single multiple-valued

variable. The logic optimizer that NOVA is intended for is ESPRESSO II. To solve the state assignment problem, NOVA uses two different approaches. The first one is to approximate the solution of the problem by modelling it as a problem of optimal encoding of inputs of a logic function. The second approach is to first model the problem as a problem of optimal encoding of both inputs and outputs (or some inputs and some outputs) of a logic function. Then, a scheme of symbolic minimization is used to partially capture the effect of the next states in the case of two-level implementations. Only the first approach is discussed in this thesis. For details on the second approach, readers can refer to [6].

To model the state assignment problem as an optimal input encoding problem, a multiple-valued logic minimization procedure, ESPRESSO-MV, is applied on a symbolic representation of the FSM (i.e. the KISS format shown in Figure 4.6). This procedure determines subsets of states that have essentially the same next state and assert the same output under some input combination. These subsets of states are called input constraints since they constrain the encoding of the present states. The goal of the state assignment is to assign each of these subsets of states to sub-cubes of a Boolean k-cube, for a minimum k. The assignment is done so that each sub-cube contains all and only the codes of the states included in the corresponding constraint. To solve this problem, several algorithms, each employing a different approach, such as exact or approximation, are developed.

Experimental results have shown that the final areas obtained by NOVA are on average 30 percent less than that obtained by random state assignment and 20 percent less than that obtained by KISS. A comparison between NOVA and MUSTANG also shows that these two produce competitive results for multi-level implementations even though NOVA targets for two-level logical implementation.

2.4 Evaluation of State Assignment

State assignment is an effective way to simplify the combinational logic expressions of the FSM. However, the area of any circuits depends on both the amount of interconnect and

the number of logic gates. After state assignment, signals are used in more places and the relationship among the logic gates becomes more complex. Consequently, the interconnect of an encoded FSM is more complex than an un-encoded FSM, increasing the routing area of the signals. For regular array structures, complicated interconnect poses even a bigger problem because the size of the array depends on the amount of interconnect. If all the signals are used basically everywhere in the array, folding of the array will be essentially impossible. Thus, using state assignment does not necessarily mean minimizing the area of the FSM. Also, as the state assignment algorithms are tailored more to optimizing the area of the FSM, the testability of the FSM are impeded [1]. It has been stated in [1] that it is easier to test an un-encoded FSM than to test an encoded one. Therefore, state assignment may not be as effective in implementing FSM as it is thought to be.

2.5 Layout Structures for FSMs

The hardware implementation of an FSM consists of two parts: the combinational part and the memory elements. The combinational part consists of the output logic and the next state logic, and is laid out using regular arrays or standard cells. The memory element which holds the actual state of the FSM is implemented using registers or flip-flops. In the following sections, several ways to implement an FSM's combinational logic and memory elements are reviewed.

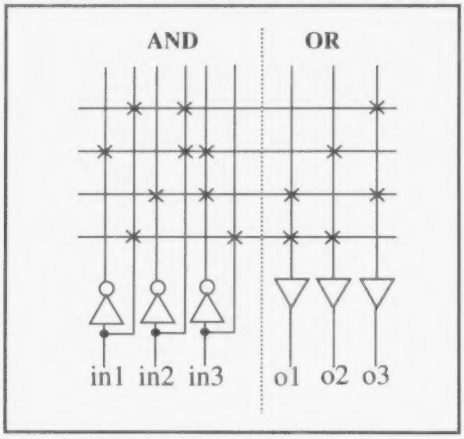
2.6 Regular Arrays

Many regular array structures have been used to lay out the combinational logic of FSMs. The advantage of regular arrays over other layout methods is that they are easy to automate. Regular array layout usually involves a gate ordering process to minimize the size of the array followed by a tiling process to generate the array. In this section, array structures like the PLA, gate matrix and Weinberger array are presented.

2.6.1 Programmable Logic Array (PLA)

Among the different kinds of regular array structures, the PLA is the most common structure used to lay out FSMs. A PLA is a regular array which is designed to implement random logic expressions in the sum-of-products form [8]. A typical PLA consists of an AND plane and an OR plane as shown in Figure 2.1.

FIGURE 2.1 Basic Structure of a PLA.



The inputs of a PLA and their complements enter the AND plane and form the necessary product terms. These product terms are then passed on to the OR plane where the outputs are generated by OR-ing the appropriate product terms together. In practice, PLAs are usually implemented using two NOR planes with inverted inputs and outputs. These two NOR planes can be implemented using pseudo-NMOS NOR gates or dynamic CMOS gates with four phase clocking. An example, a sum-of-products form implemented using two level of NOR gates is shown as follows:

$$
\begin{aligned}
out &= ab + c\bar{d} \\
\overline{out} &= \overline{ab + c\bar{d}} \\
&= \overline{\bar{a}\bar{b}} + \overline{\bar{c}d} \\
&= \overline{\bar{a}\bar{b}} + \overline{\bar{c}d}
\end{aligned}
$$

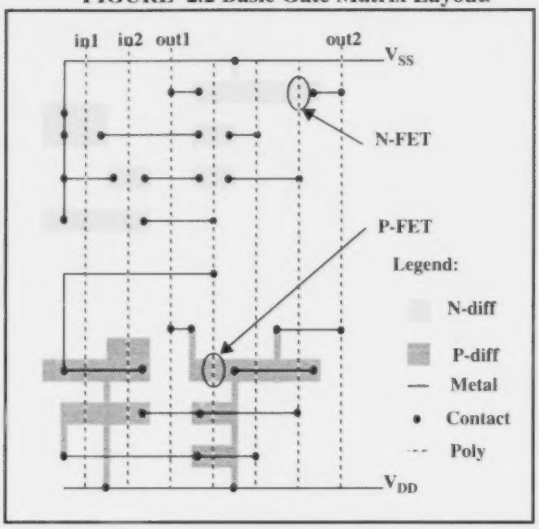
Much work has been done on trying to minimize the size of PLAs, or to increase the device-packing density of PLAs. One of the methods for minimizing PLA size is to reduce the number of product terms. This can be done by developing a better state assignment

algorithm. Another well known technique which can minimize the size of the PLA is PLA folding. As discussed in [9], PLA folding involves merging the signals on two or more PLA rows or columns into a single row or column. Simple column folding is the simplest folding, and involves merging pairs of columns into a single column. Other more complex foldings, for example, are constrained folding and multiple column folding. In constrained folding, the locations of inputs and outputs are restricted due to routing problems. In multiple column folding, the PLA's inputs and outputs enter or exit the PLA horizontally through the left and the right of the PLA instead of vertically through the top and the bottom of the PLA. Arranging the inputs and outputs this way allows two or more columns to be merged into a single column.

2.6.2 Gate Matrix

Gate matrix is a multi-level logic regular array structure. As presented in [10], gate matrix simplifies and unifies the layout procedure by using an orderly structure: a matrix composed of intersecting columns and rows. Gate matrix is a static CMOS layout structure. All the N-transistors are located in half of a matrix, and all the P-transistors are located in the other half. The columns which serve the dual role of transistor gate and interconnections are implemented with polysilicon. The rows, which are diffusion, intersect with the columns to form transistors. As an example, a sample gate matrix layout is shown in Figure 2.2.

FIGURE 2.2 Basic Gate Matrix Layout.



The advantages of the gate matrix layout are that it can be readily transformed into its final mask artwork, and its design can be updated to new design rules easily. Moreover, gate matrix has a higher device-packing density than the PLA.

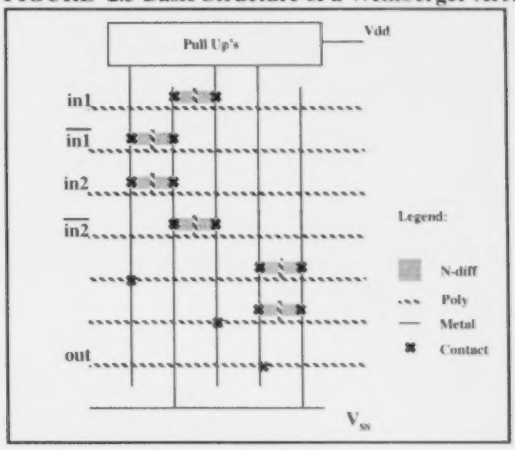
Various techniques have been developed for minimizing the gate matrix. One of the techniques discussed in [11] is gate matrix partitioning. In this technique, a gate matrix is divided into two gate matrices, an upper and a lower gate matrix, by partitioning the nets in the original gate matrix into two groups. Then, the interconnecting gates between the two matrices (i.e. gates that intersect nets in both sub-matrices) are placed in the same columns of the upper and the lower matrices so that no extra routing area is needed. The gates that are local to one gate matrix can share columns with the other gates that are local to the other gate matrix. Thus, the overall width of the original gate matrix is reduced. This process can be recursively applied to the partitioned gate matrices until the circumscribing rectangle has the best combination of area and aspect ratio.

2.6.3 Weinberger Array

Another multi-level regular array layout for FSMs is the Weinberger array. The structure of a Weinberger array is very similar to that of a PLA [12]. The difference is that a

Weinberger array only consists of one plane. A typical NMOS Weinberger array with a NOR plane is shown in Figure 2.3.

FIGURE 2.3 Basic Structure of a Weinberger Array.



In a Weinberger array, the columns (gates) run in metal, and the rows (signal lines) run in polysilicon. Inputs are applied to one end of the signal lines. To form a transistor, a horizontal diffusion tab is placed connecting the ground line and the metal gate column. Then, a vertical polysilicon tab is extended out from a signal line. A transistor gate is formed by running the tab of polysilicon across the tab of diffusion. By connecting the outputs of the gates to the polysilicon signal lines via contact cuts, these outputs are carried to the inputs of other columns.

The advantage of the Weinberger array is that multi-level functions can be expressed easily. The disadvantage of the Weinberger array is that it tends to take an awkward shape with its width being much larger than its height. This is because gate pitch (i.e. column width) of a Weinberger array layout is usually larger than the row pitch.

Several techniques have been developed to increase the Weinberger array's device-packing density and to improve its aspect ratio. One of the techniques discussed in [12] is the column folding technique. Assuming NMOS logic is used, the P-type pull-up transistors are allowed to be placed both on top and at the bottom of a column. This means that two columns or gates in the unfolded array are placed in one column in the folded

array wherever possible. This results in a saving of area, as well as reducing the width of the Weinberger array. Further area saving can also be obtained with column folding followed by row compaction.

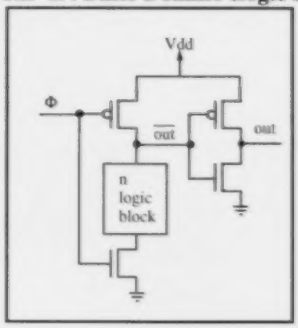
2.6.4 Layout Structure Realization

Both the PLA and Weinberger array can be realized in different types of logic. The most common one is pseudo-NMOS logic since it requires less area compared with static CMOS. This is because the static CMOS logic has both P-type and N-type transistor trees, whereas, the pseudo-NMOS logic only has an N-type transistor tree. However, the price that is paid by pseudo-NMOS is static DC power dissipation because the P-type pull-up transistor is always on.

One way to solve this problem is to use dynamic logic instead of pseudo-NMOS. A dynamic logic gate is a clocked logic gate. It consists of a P-type pull up transistor and a pull down N-transistor logic tree, and its output value is stored as an electric charge at the output node. Since dynamic logic gates are controlled by one or more clock signals, either the pull-up (pre-charge) or pull down path (discharge) can be on at one time. Thus, dynamic logic does not suffer from the static power dissipation problem of pseudo-NMOS, and still takes less space than static CMOS logic. Two types of dynamic logic gates are reviewed in the following sections.

2.6.4.1 Domino Logic

FIGURE 2.4 Basic Domino Logic Structure

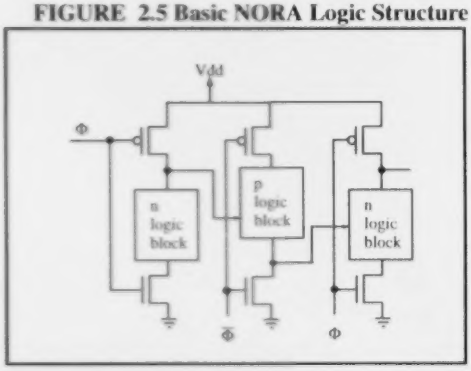


The basic domino logic structure is shown in Figure 2.4. It consists of a clocked CMOS logic gate and a static CMOS inverter. The operation of a domino logic gate involves two phases: precharge and evaluation. During the precharge phase, clock signal Φ is low. The P-transistor is on and the N-transistor block is off. This results in node $\overline{out}$ being charged up to a logic '1'. One should notice that, without the inverter, the output of the following gate may be momentarily discharged by the precharged node during evaluation. Adding the inverter prevents it from happening. During the evaluation phase, Φ is high. The P-transistor is off, and node $\overline{out}$ is conditionally pulled down by the N-transistor block.

The advantage of this logic gate is that it allows us to precharge or evaluate a cascaded set of clocked CMOS logic gates using a single clock. The disadvantages of domino logic gate are that the extra static inverter takes up space, and it makes laying out the logic gate more difficult. Also, since every cascaded stage is buffered, domino logic can only implement non-inverting logic expressions. Further more, domino logic, like any other dynamic logic gate, also suffers the charge sharing problem. Charge sharing means that the electric charge stored in the output node are redistributed among the parasitic capacitors in the logic gate. Depending on the sizes of the parasitic capacitors, this change in electric charge may affect the output value of the gate.

2.6.4.2 NORA

FIGURE 2.5 Basic NORA Logic Structure



NORA logic is a refinement of domino logic. Like domino logic, it allows us to precharge or evaluate a cascaded set of clocked CMOS logic gates using a single clock. However,

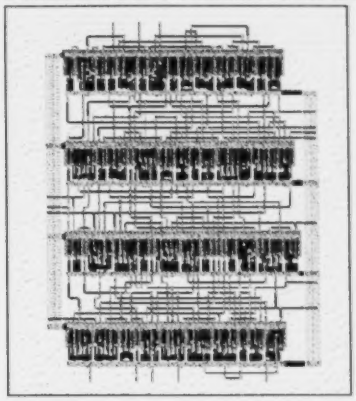
instead of adding an extra inverter at the end of each clocked CMOS gate, it uses alternate P-type and N-type transistor logic blocks [13]. The operation of a NORA logic chain involves the precharge, and the evaluation phases. During the precharge phase, the clock signal Φ is low. Thus, the output nodes of all the NORA gates with an N-type logic block are pulled up to a logic '1' by the P-type transistors, and the P-type transistors in the succeeding stage are turned off. On the other hand, the output nodes of all the NORA gates with a P-type logic block are pulled down to a logic '0' by the N-type transistors. Thus, the N-type transistors in the succeeding stage are turned off. During the evaluation phase, Φ is high. all gates that have a P-type logic blocks are conditionally pulled up to logic '1's while those that have an N-type logic blocks are conditionally pulled down to logic '0's.

The advantage of NORA logic is that it allows us to precharge or evaluate a cascaded set of clocked CMOS logic gates using a single clock. Some of NORA's disadvantages are that it has a slower response because of the P-type logic blocks, and that it also suffers from charge sharing.

2.7 Standard Cells

In addition to using regular arrays, FSMs can also be implemented using a set of predefined logic circuit cells. This layout method is called standard cell layout. In this method, the layout consists of two parts [10]. The first part is the predefined logic which is contained within cells that are grouped together in rows. These cells all have the same standard height and are implemented using static CMOS logic. The second part is the wiring channels between rows of cells. The wiring channel consists of two types of isolated conductors and serves as interconnection among the logic cells. A typical standard cell layout is shown in Figure 2.6. In order to obtain a more area efficient standard cell layout, various placement and routing algorithms have been proposed. In fact, this has been one of the most active research areas in computer-aided design.

FIGURE 2.6 Basic Standard Cells Layout.



The advantage of using standard cells to implement FSMs is that it is easier to develop the layout. However, it usually gives larger FSMs because it uses static CMOS logic. The following table gives an area comparison between the combinational parts of several FSMs provided by MCNC implemented using PLA and standard cells.

TABLE 2.1 Standard Cells Layout vs. PLA

FSM Name	#. of States	#. of Inputs	#. of Outputs	#. of Edges	Standard Cells *	PLA**	Standard Cells / PLA
opus	10	5	6	22	230880	158610	1.46
ex4	14	6	9	21	258280	148512	1.74
dk512	15	1	3	30	175960	101556	1.73
planet	48	7	19	115	2395232	704088	3.40
keyb	19	7	2	170	767536	344256	2.23

* Implemented using NOVA as the state assignment tool and MIS as the logic optimizer.
** Implemented using NOVA as the state assignment tool and ESPRESSO as the logic optimizer.

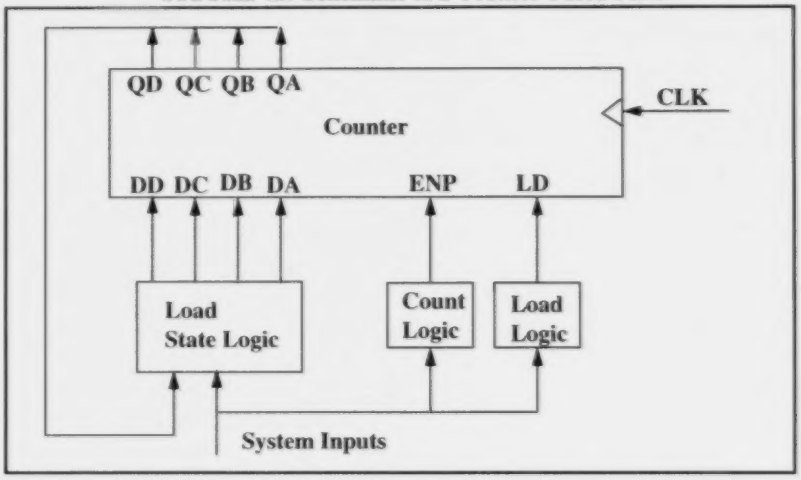
In Table 2.1, the given area is in the design unit used by a design synthesis system called OCTTOOLS. On average, the area of the standard cell implementation is about twice of that of the PLAs. Thus, we can see that, in general, standard cell implementation of an FSM gives larger area than a PLA implementation.

2.8 Counter-Based FSMs

The conventional way to implement the memory elements of an FSM is to use flip-flops such as the D flip-flop or the JK flip-flop. However, other sequential devices such as

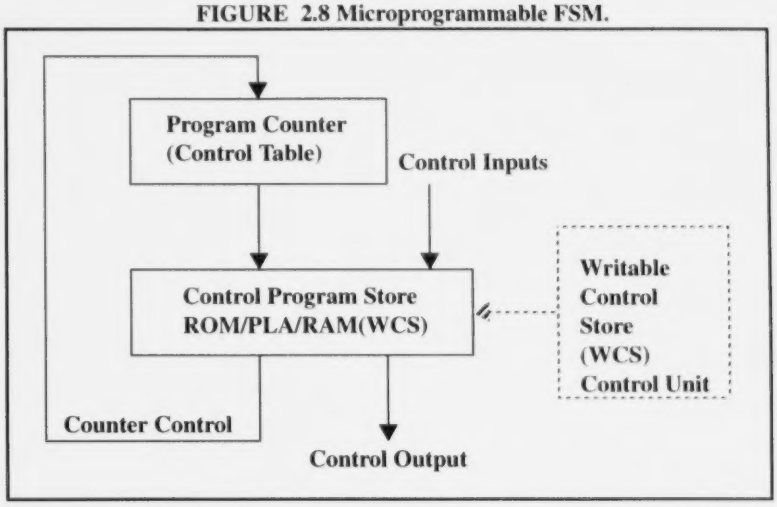
loadable counters can also be used as the memory elements. The counting operation of a loadable counter is closely allied to the process of state-to-state transition [8]. For example, the state transition of going from state 000 to 001 is merely a simple count operation. The state transition of going, say, from state 000 to 100 can be done using a load operation. If a state assignment is able to maximize the number of state transitions that can be implemented by incrementing the counter, the counter based FSM can be implemented efficiently in terms of area [14]. This is because a compactly laid out counter can replace much of the next state logic implemented in a less compact structure (e.g. PLA). The organization of a counter-based FSM is shown in Figure 2.7.

FIGURE 2.7 Schematic of a Counter-Based FSM.



2.9 Microprogrammable FSM

FIGURE 2.8 Microprogrammable FSM.



A microprogrammable FSM is one in which a designer can program the device's operation by writing instructions into the device's memory. Also, it is modular so that it can be expanded to an n-state FSM. As shown in Figure 2.8, a microprogrammable FSM usually consists of a counter unit, which allows the FSM to increment or branch to the next instruction, and a control store unit which stores the microprogram that controls how the FSM works. The control store unit is usually implemented using RAM or ROM. In the latter case, the FSM is called a microprogrammed FSM instead of microprogrammable. According to [8], a microprogrammable FSM has the following features:

1. It is able to start processing a stored instruction program starting from an arbitrary location in memory.
2. It can process instructions conditionally or unconditionally stepping in sequence through the memory.
3. It has the inherent capability of processing the next instruction or "branching" to another location in a stored instruction list conditionally or unconditionally.

The principal advantage of this method over all of the previously discussed methods is that the design of the FSM can be defined at a later stage of the circuit design and can be easily modified (i.e. FSM with a writable control store unit). Another advantage is that it

does not require state assignment. One of the disadvantages of this method is that it usually gives a larger and less compact FSM. Another disadvantage is that since each state transition involves executing an instruction in the memory, the speed of the FSM is impeded.

2.10 Evaluation of Current Layout Structures

As presented in the previous sections, there are numerous ways to implement an FSM; however, all seem to leave room for improvement. For the regular array structures, all require additional memory devices to hold the FSM state. These memory devices not only take up extra area, but also increase the routing complexity. Even though there are many ways to increase the device-packing densities of these arrays, the device-packing densities are still in general not very high. Thus, the resulting layout is not area efficient. As for the standard cell layout, since it uses static CMOS gates, it gives a larger FSM layout. The use of counters is an interesting way to implement an FSM. Unfortunately, there are not many state assignment algorithms that are devoted to this type of FSM implementation. Moreover, the effectiveness of this method depends heavily on the nature of the FSM being implemented. In many cases, although the amount of next state logic is reduced, the amount of area saving may be offset by the size of the counter unit. The microprogrammable FSM is an efficient method to implement an FSM if flexibility is more important than minimizing the area.

2.11 Summary

Many different ways of implementing FSMs are reviewed in this chapter. However, all of them have limitations. State assignment seems to be an effective way to simplify the logic expressions of the combinational logic of the FSM. However, the final size of the FSM depends on how good the state assignment algorithm works with the logic optimizer. Also, from a testing point of view, testing an encoded FSM requires more work than testing an un-encoded FSM.

We believe that there must be a better way to implement FSM. In the next chapter, a new method to implement FSM using one-hot state assignment (i.e. un-encoded FSM) will be presented.

CHAPTER 3

New FSM Implementation

3.1 Problem Definition

As mentioned in Chapter 2, an FSM is usually implemented in two steps. The first step involves logic minimization using state assignment and a logic optimizer. The second step involves laying out the combinational logic of the FSM in an array structure such as PLA, and using flip-flops as memory elements. Implementing an FSM in this way has three main disadvantages. First, it requires a state assignment stage. This makes the size of the final FSM layout depend very much on the efficiency of the state assignment algorithm used. Second, the testability of a encoded FSM is decreased. Third, since memory elements are attached to the array, a certain amount of area will be consumed in routing and trying to align the array with the memory elements.

In this project, our objective is to come up with a better implementation method such that the area of the FSM is minimized without sacrificing its testability. To do this, we have decided on using one-hot state assignment in our project. One-hot state assignment means that every state in an FSM is assigned one state variable, and only one state variable is true at one time. In other words, we have an un-encoded FSM. We choose to use one-hot encoding because we believe that it will give us a smaller FSM and better testability. Examining flow graphs of several sample FSMs, we find that most of the inputs are only used in a few states which are usually close together. Thus, using an un-encoded FSM, the length of the input signal nets can be reduced, and, hence, the area of the FSM reduced.

In order to develop a new layout method tailored for FSMs, we have come up with a list of specifications that our new method should accomplish. This specification list will guide us through the development of our new method. The specification list is as follows:

1. one-hot state assignment.

One-hot state assignment is used in this project in order to minimize the size of the FSM and maintain testability.

2. dynamic logic or pseudo-NMOS logic.

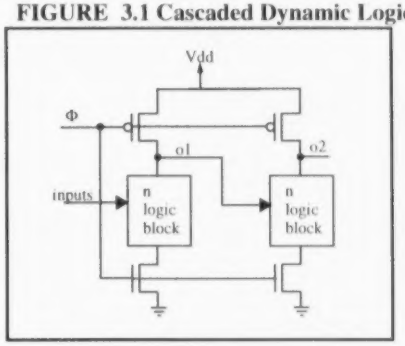
   In order to minimize the final area of the FSM layout, we choose to use dynamic or pseudo-NMOS logic rather than static CMOS logic. Dynamic logic has the advantage over pseudo-NMOS logic of having no static power dissipation. However, it is more difficult to design.

3. parallel transistor logic wherever possible.

   In sizing transistors, if a gate has transistors in series, the channel widths of the transistors must be increased according to their number in order to maintain speed performance. If a gate has many transistors in series, the size of the gates will become fairly large. Therefore, it is best to use parallel transistor logic, such as the NOR gate in pseudo-NMOS logic, in our new FSM layout.

4. must be free of dynamic races and charge sharing problems.

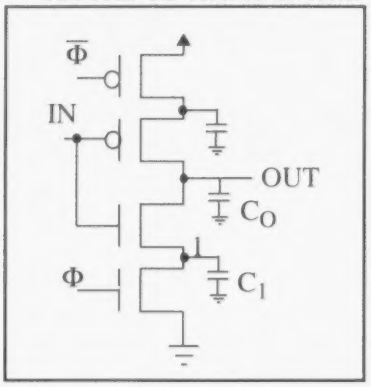
FIGURE 3.1 Cascaded Dynamic Logic



A dynamic race occurs when basic dynamic logic gates are cascaded together. In Figure 3.1, during the pre-charge phase, o1 and o2 are charged up to Vdd. During the evaluation phase, if o1 is to be discharged by the N logic tree, o2 should stay high all the time. However, due to the pull down time of o1, o2 may be momentarily discharged by the pre-charged o1. This situation is called a dynamic race.

The charge sharing or charge re-distribution problems are normally related to using dynamic logic gates.

FIGURE 3.2 Clocked Inverter.



A clocked inverter along with some of its parasitic capacitances are shown in Figure 3.2. Consider the situation when Φ is high and IN is low, OUT is charged up to high while node 1 is discharged to low. After that, when Φ is low and IN is high, the charge from C_O will flow to C_1. Depending of the ratio between C_O and C_1, the amount of charge sharing among these two parasitic capacitances may affect the output voltage.

5. at most two clock phases.

   The data path of a digital system requires a minimum of two clock phases: one which controls the functional units, and another which controls the registers. If our FSM requires more than two clock phases to operate, the two-phase-clocking data path will have to wait for additional clock phases before it can start its next cycle. Thus, in order for our FSM be able to work with the data path without slowing it down, our FSM must be able to operate using two clock phases.

6. have memory elements embedded within FSM.

   Instead of having the memory elements being attached to the combinational logic block of the FSM, the memory elements will be embedded within the combinational logic block. Laying out the FSM in this way gives a more compact structure. Also, the placement stage usually needed to fit these two blocks together can be eliminated.

7. must be easy to automate.

   Nobody wants to spend hours laying out an FSM manually. Thus, a very important feature of this new FSM layout method is that it must be easy to automate. This means that once the user gives the FSM layout generator a high level description of the FSM, the generator should produce the layout of the FSM with no or minimum attention by the user.

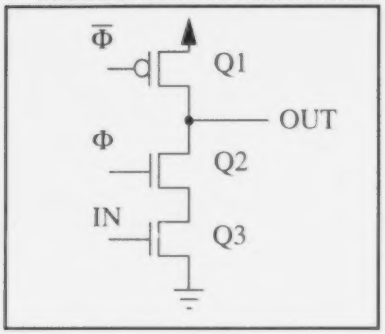
In order to come up with a new FSM layout method that satisfies all the above specifications, several different new layout ideas were attempted. Among these new ideas, a method called CLORA FSM has been chosen to be automated because it gave good preliminary results. In the rest of this chapter, this new FSM layout method, CLORA FSM, will be described in detail.

3.2 Clocked Ratioed Logic Gate

During this project, we have experimented with dynamic logic such as NORA and Domino logic in order to have the memory device embedded within the FSM. However, none of these satisfies our need. Domino logic requires an extra inverter for every cascaded stage. These extra inverters not only take up extra space, they also complicate the way the FSM is laid out. Also, they limit the Domino logic to being only capable of implementing non-inverting gates. NORA logic requires alternate use of N-type and P-type transistors. This makes it very difficult to come up with an FSM logic realization that uses only parallel transistors.

Our new layout method is called CLORA FSM because it uses a new type of design style, called CLORA, as its basic building block. CLORA stands for CLOcked RAtioed and was developed as part of this research project.

FIGURE 3.3 A CLORA Inverter.



3.2.1 Operation of CLORA Logic

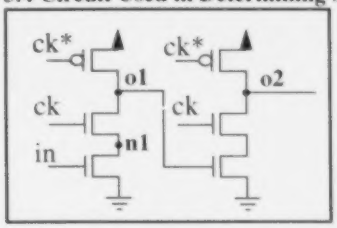
A CLORA inverter gate is shown in Figure 3.3. A CLORA gate is essentially a clocked pseudo-NMOS gate. The operation of a CLORA gate is straight forward. When the clock signal Φ is low, transistors Q1 and Q2 are off. This means that the output node OUT is in a high impedance state. During this time, the output value of the CLORA gate is unchanged, and the value is held by the charge on the output node. Thus, the CLORA gates behaves the same as a dynamic logic gate. When Φ is high, transistors Q1 and Q2 are on. During this time, the CLORA gate behaves the same as a ratioed logic gate, and the output node OUT is evaluated. For a CLORA gate to function properly, the rising edge of the complementary clock signal, $\overline{\Phi}$, must lead the falling edge of the clock signal, Φ (i.e. Q1 must be turned off before Q2). If this condition is not met, Q2 may be turned off before Q1, and the output value evaluated when Q2 is on may be pulled up to high by Q1. Thus, the timing between Φ, and $\overline{\Phi}$ is very important for a CLORA gate.

3.2.2 Designing the CLORA Logic

Since a CLORA gate is a clocked ratioed logic gate, the voltage level of the output node depends on the aspect ratio of the P-type and the N-type transistors. In order to design a type of CLORA gate that will work with CMOS logic, we have to determine the aspect ratio of the transistors in the CLORA gate experimentally. Using the CLORA inverter gate shown in Figure 3.3, an experiment is performed as follows:

1. Connect the output of one CLORA inverter to the input to another CLORA inverter (Figure 3.4).

FIGURE 3.4 Circuit Used in Determining Aspect Ratio.



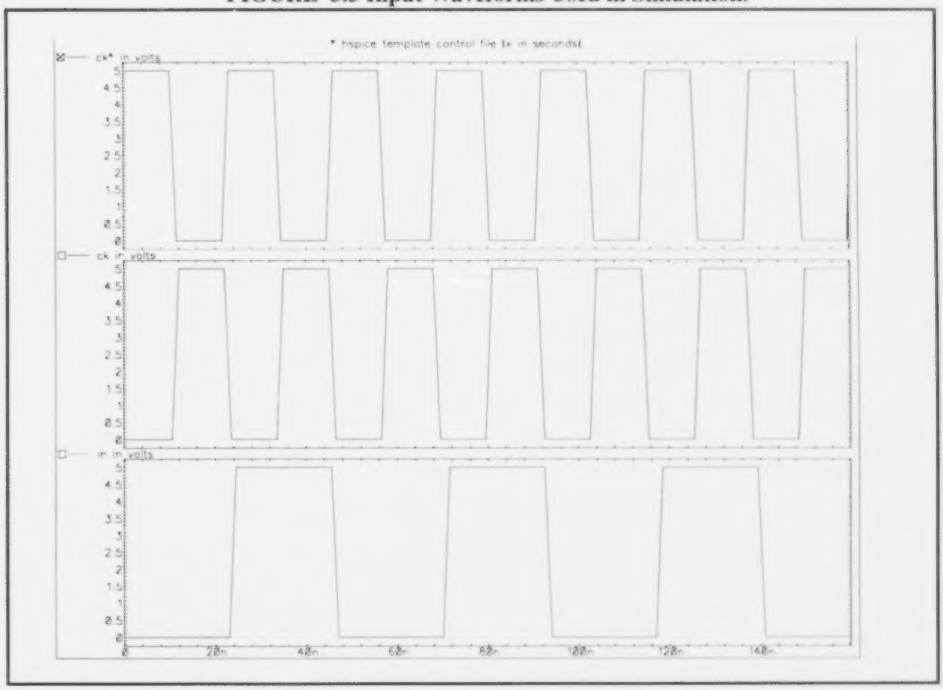
The second CLORA inverter is used as a load to the first CLORA inverter. We are interested in the timing and the transfer characteristics of the output of the first CLORA inverter (i.e. node 'o1').

2. Keep the channel widths of all the Q1's, Q2's, and Q3's to 4.0 design microns (CMOS4S technology).

3. Keep the channel length of Q2's, and Q3's to 1.5 design microns.

4. Simulate the circuit with the input waveforms shown in Figure 3.5 using hspice to obtain the transfer and the timing characteristics of the CLORA inverter.

FIGURE 3.5 Input Waveforms Used in Simulation.

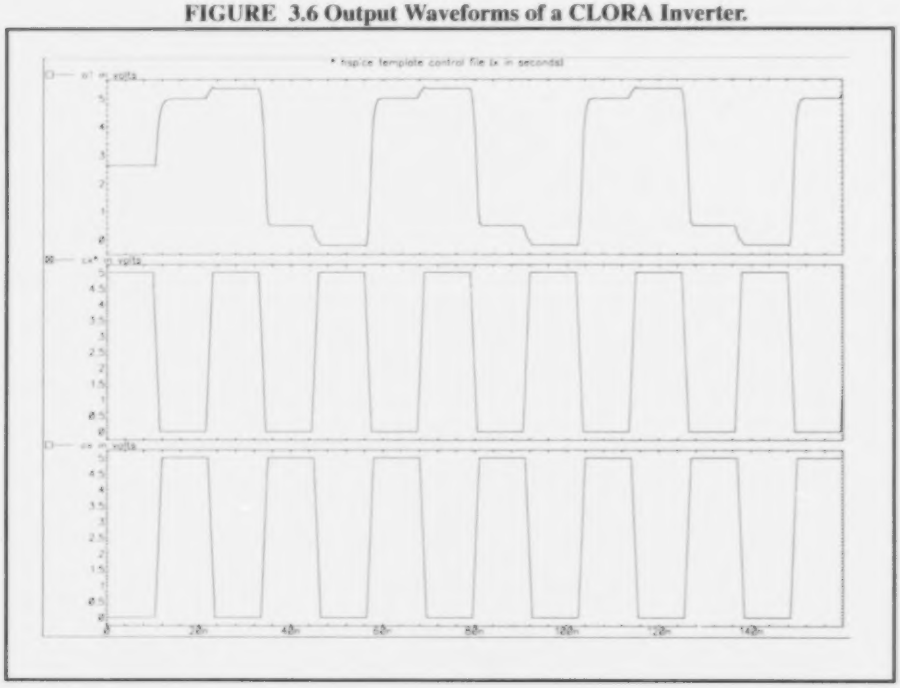


5. Change the channel length of Q1 to give different transfer and timing characteristics of the CLORA inverter.

6. Using noise margins and speed as our criteria, examine the transfer and the timing characteristics, and choose the aspect ratio that has a good overall performance. As a guide line, a noise margin of at least 0.7 volts is required.

For example, increasing the channel length of Q1 increases the pull up resistance of the CLORA gate. This increases the time it takes to pull the output node to a logic '1'. Also, the V_{OL} of the CLORA gate is lower.

FIGURE 3.6 Output Waveforms of a CLORA Inverter.



The waveform of 'o1' is shown in Figure 3.6. When 'o1' is a logic '0', a voltage drop occurs after Q1 is off (i.e. ck* is high). This voltage drop is caused by charge sharing. When 'in' is high, node 'n1' is discharged. After evaluation and Q1 is off, there is a certain delay before Q2 is off. Thus, charge sharing occurs between 'o1' and 'n1', and that pulls the voltage of 'o1' further down. Since this charge sharing makes a 'low' voltage even lower, this charge sharing does not affect the proper functioning of the circuit.

When 'o1' is a logic '1', a voltage increase occurs after Q1 is off. This is caused by clock feed-through. When ck* switches to a logic '1', it forces the charge remaining in the Q1's channel out of the channel. Some of the charge flows to 'o1', and causes the increase in voltage. Again, this clock feed-through does not affect the proper functioning of the circuit since it makes a 'high' voltage even higher.

The transfer and timing characteristics of the CLORA inverter gate with various aspect ratios is shown in Table 3.1. In this table, the channel length is in design microns. The t_r value is the rise time of o1. This is obtained by varying the rise time of the input signals until a point when both the inputs and the output have similar rise time. The noise margins NM_L and NM_H are defined as:

$$NM_L = V_{IL} - V_{OL}$$

$$NM_H = V_{OH} - V_{IH}$$

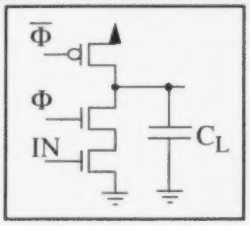
TABLE 3.1 Transfer Characteristic for a CLORA Gate with Different Aspect Ratios.

Aspect Ratio (L_{Q2}:L_{Q1})	t_r (ns)	V_{IL}	V_{IH}	V_{OL}	V_{OH}	NM_L	NM_H
		(V)					
1.5 : 2.0	1.35	1.75	4.85	0.86	5.00	0.89	0.15
1.5 : 2.5	1.25	1.75	4.60	0.64	5.00	1.11	0.40
1.5 : 3.0	1.55	1.75	4.25	0.50	5.00	1.25	0.75
1.5 : 3.5	1.9	1.75	3.80	0.42	5.00	1.33	1.20

By examining Table 3.1, we find that the NM_H given by the first two aspect ratios are too low. Thus, we cannot use the first two aspect ratios. On the other hand, both the third and the fourth aspect ratios give good noise margins. In order to have a good noise margins without sacrificing too much on area and speed performance, the aspect ratio of the CLORA NOR gate used in our FSM layout is chosen to be 1.5 : 3.0. Thus, the channel length of Q1 and Q2 are 3.0 design microns and 1.5 design microns respectively.

3.2.3 Timing Characteristic of a CLORA Inverter

FIGURE 3.7 CLORA inverter with C_L.



To further investigate the timing characteristic of the CLORA gate with the 1.5 : 3.0 aspect ratio, we simulated the CLORA inverter shown in Figure 3.7 with two different C_L values. The first C_L value is 5.3fF, and the second one is 256fF. In both cases, we measured the time it took for the output to go from the 10% point to the 90% point, t_r, and the time it took for the output to go from the 90% point to the 10% point, t_f. Then, assuming that our timing functions have the form

$$t_X = t_{X0} + m_X \bullet C_L$$

where t_X represents either t_r or t_f in ns and C_L is in fF, we came up with the timing functions for both t_r and t_f are

$$t_r = 1.04 + 3.11e\text{-}2 \bullet C_L;$$
$$t_f = 1.09 + 2.11e\text{-}2 \bullet C_L;$$

3.2.4 Pros and Cons of CLORA Logic

Since the CLORA gate is partially dynamic and partially ratioed logic, it inherits many advantages and disadvantages from these types of logic. These are as follows:

Advantages:

1. Once the output is evaluated, a CLORA gate can retain its output value until the next clock cycle.
2. The CLORA gate can be cascaded easily. Unlike Domino logic, no extra inverter is required to connect the cascaded stages because the output nodes are precharged at the same time as they are evaluated.

Disadvantages:

1. Since the output value is stored as an electric charge, if the output value is not read or refreshed fast enough, the value might change. Thus, the CLORA gate must be clocked at a reasonable speed.
2. Since the pre-charge and the evaluation phases occur at the same time, we have to make sure that Q1 turns off before Q2.

3.3 FSM Realization

One can use CLORA logic to implement any synchronous logic circuit. In this project, we implemented FSMs using CLORA NOR gates as the basic building block. In the next section, we will describe how to use CLORA NOR gates to implement un-encoded FSMs.

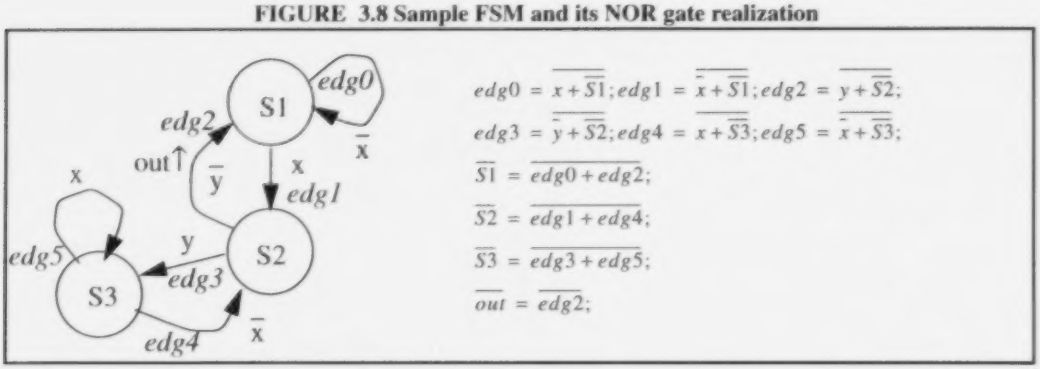
3.3.1 Un-encoded FSM realization

In order to aid the reader in understanding how an un-encoded FSM is realized, a type of directed graph called a state diagram is presented first. An example of a state diagram is shown in Figure 3.8. There are three nodes in the graph; each of which represents a state in the FSM. Also, there are six edges in the graph; each of which corresponds to the entrance condition (i.e. edge pointing toward a state) or the exit condition (i.e. edge pointing outward from a state) of a state. Since an FSM can only be in one state at one time, and all the exit conditions of a state are mutually exclusive, only one entrance/exit condition can be true at one time. Using one-hot state assignment, each state in the un-encoded FSM is assigned a unique state variable. Depending on the evaluation of the exit or entrance conditions of a state, the state variable can be either '0' (i.e. the FSM is not in that state), or '1' (i.e. the FSM is in that state).

In our implementation, we have decided to use the entrance conditions of a state to determine the value of a state variable. Thus, a state variable has a '1' value if and only if one of its entrance conditions is true. Otherwise, the state variable has a '0' value. To differentiate the CLORA gates that are used to evaluate the entrance conditions from those

that are used to evaluate the state variables, we call the former edge gates, and the latter, state gates. Also, the CLORA gates that are used to evaluate the outputs of the FSM are referred to as output gates. As an example, the NOR gate realization of the FSM described in the state diagram is also shown in Figure 3.8.

FIGURE 3.8 Sample FSM and its NOR gate realization



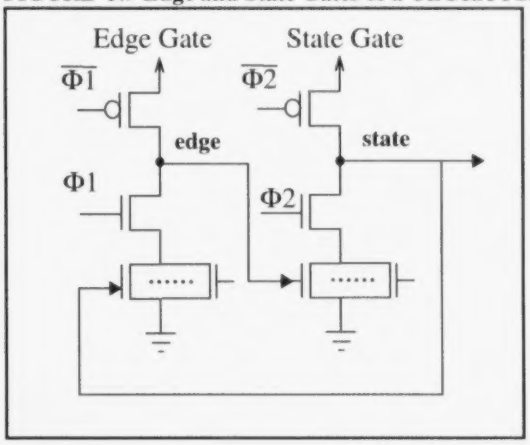
Using CLORA NOR gates, the entrance conditions of state S1, S1 • $\bar{x}$ and S2 • $\bar{y}$, are evaluated by the two edge gates, edg0 and edg2, respectively. Then, the complementary signal, $\overline{S1}$, is evaluated by NORing the outputs of edg0 and edg2 together. As shown in the above FSM logic expressions, only the complementary signals of the state variables are needed in evaluating the entrance conditions of the FSM. Thus, no extra inverters are needed to obtain the true value of the state variables.

In our un-encoded FSM realization, two methods are used to implement the outputs of the FSM. The first method is to NOR all the edges that assert the output (the so-called 1-edges) together as shown in the Figure 3.8. This gives us the complementary value of the output. Then, an inverter is used to obtain the true output value. The second method to implement the output is to NOR all the edges that de-assert the output (the so-called 0-edges) together. This gives us the true value of the output. Notice that the second method only works for un-encoded FSM. This is because with an un-encoded FSM, only one edge can have a logic 1 at one time. When a 1-edge of an output is true, it is obvious that all of the 0-edges of that output are false. Thus, the output will have a true value. The way to determine which method to apply to an output will be discussed in detail later on in the thesis.

3.4 The Operation of a CLORA FSM

For the CLORA FSM to operate properly, the time at which the edge gates, state gates and the output gates are evaluated is very important. Since the outputs of the edge gates are inputs to both the state gates and the output gates, the outputs of the edge gates must be ready and remain unchanged when the other two types of gates are being evaluated. On the other hand, since the external inputs of the FSM and the outputs of the state gates are inputs to the edge gates, these signals have to be ready and remain unchanged when the edge gates are being evaluated. In order to ensure that these timing conditions are met, two non-overlapping clocks, Φ1 and Φ2, are used in implementing a CLORA FSM.

FIGURE 3.9 Edge and State Gates of a CLORA FSM.



With Φ1 controlling the edge gates and Φ2 controlling the state and output gates, the CLORA FSM operates in the following way:

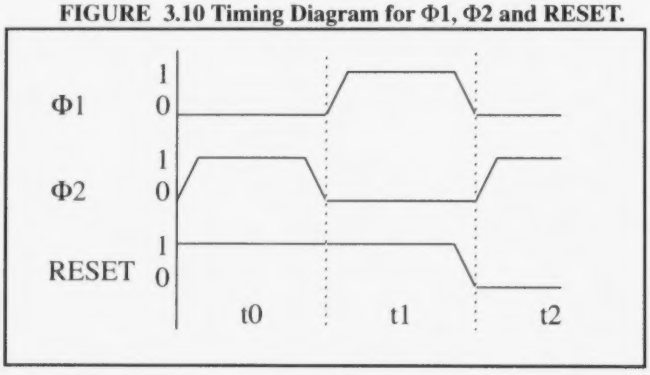
- When Φ1 is high and Φ2 is low, all the edge gates are evaluated. The state gates and the output gates are in a high impedance state, and their output values remain unchanged. This feature allows the FSM to remember which state it is in.
- When Φ2 is high and Φ1 is low, all the state gates and output gates are evaluated. At the same time, the edge gates are in a high impedance state, and their output values stay unchanged.

Since only two non-overlapping clocks are used in a CLORA FSM, the CLORA FSM will not slow down the data paths that require only two clock phases to operate.

3.5 Reset Mechanism

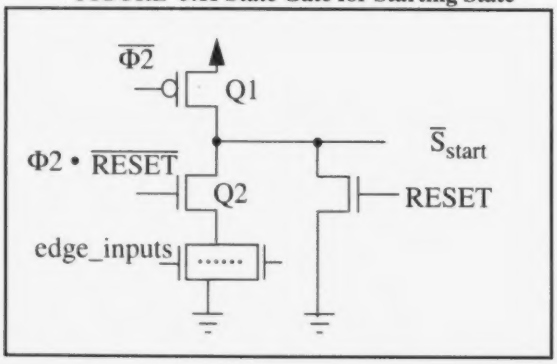
Since one-hot state assignment is used in our FSM implementation, we need a way to reset all the state variables, except the starting state variable, to false (i.e. set the state gate outputs to '1'), during initialization or reset. In a CLORA FSM, a reset signal, RESET, with the timing relationship with Φ1 and Φ2 as shown in Figure 3.10 is used to reset the FSM to the starting state.

FIGURE 3.10 Timing Diagram for Φ1, Φ2 and RESET.



To incorporate the reset mechanism into the FSM, a slight modification is made to the CLORA FSM. First, the Q2's of the state gates and the output gates are driven by Φ2 • $\overline{\text{RESET}}$, instead of by Φ2. Moreover, the state gate that evaluates the complement of the starting state variable (i.e. starting state gate) is modified as shown in Figure 3.11.

FIGURE 3.11 State Gate for Starting State



The operation of this reset mechanism is as follows:

- t0:

  All the edge gate outputs are in the high impedance state. All the state gate outputs, except for the starting state gate, are pre-charged high because their Q1's are on and Q2's are off. Thus, all the state variables, except for the starting state are false. In the case of the starting state gate, its output is pulled low by the RESET signal. Thus, the starting variable is true.

- t1:

  The reset cycle has completed; however, RESET has to stay on until the starting state gate's Q1 is turned off. This is to ensure that the starting state gate's output will not be pulled high by Q1. Thus, RESET stays on in this time. Also, in this time, all the edge gates are evaluated as normal.

- t2:

  After the reset period, the RESET signal stays low until the next reset period.

3.6 Pros and Cons of the CLORA FSM

Compared with the PLA implementation, there are some advantages and disadvantages with using the CLORA design style to implement an FSM, and they are as follows:

Advantages:

1. The memory elements of the FSM are embedded within the combinational logic.
2. The reset mechanism of the CLORA FSM is straight forward.
3. CLORA FSM has less power dissipation than a pseudo-NMOS PLA. In a pseudo-NMOS PLA, the P-type pull up transistors are always on; whereas, in a CLORA FSM, the Q1 and Q2 transistors are only on during the evaluation phase.

Disadvantage:

- The layout of the FSM has to be designed very carefully to make sure that for every CLORA gate, the Q1 transistor turns off before the Q2 transistor.

3.7 Layout Structure of a CLORA FSM

In order that our layout structure be easily automated, we have chosen to use a regular array structure. To help reader in understanding how different CLORA gates are laid out in the array, two terms are used to describe two main parts of a CLORA NOR gate (i.e. the basic building block of the CLORA FSM). A CLORA NOR gate is shown in Figure 3.12. The "input section" of the NOR gate refers to the parallel N-transistor structure. The "output section" of the NOR gate refers to the section between Q1 and Q2 of the gate (i.e. the output node).

FIGURE 3.12 A CLORA NOR Gate

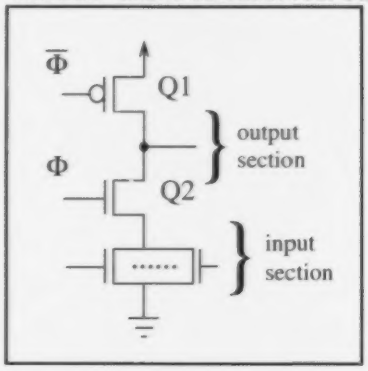
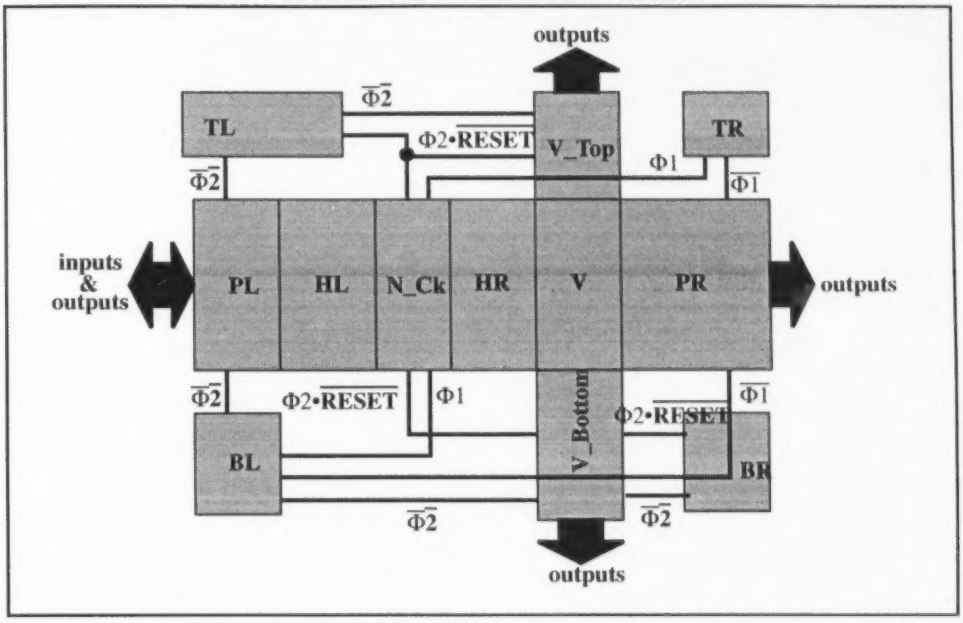


FIGURE 3.13 A CLORA FSM Structure.

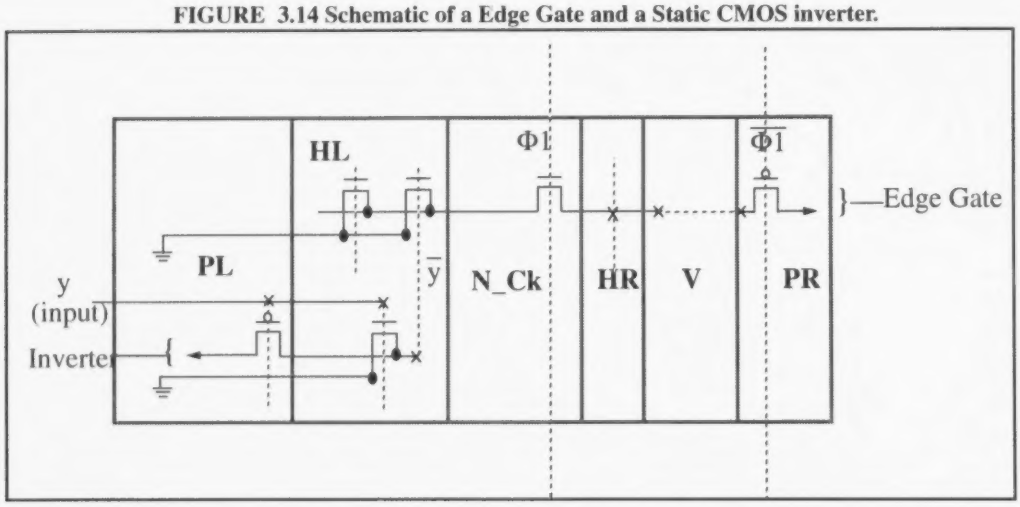


As shown in Figure 3.13, the CLORA array consists of N-transistor regions, P-transistor regions and clock buffer regions. For N-transistor regions, there are the HL plane, the HR plane, the N_Ck plane and the V plane. The HL and HR planes contain input and output sections of CLORA gates that are laid out horizontally. The N_Ck plane contains the Q2's of all the horizontal and T-shaped CLORA gates. The V plane contains input sections of the CLORA gates that are laid out vertically or T-shaped. For P transistor regions, there are the two clock planes, PL, and PR. The PR plane contains the Q1's of all the edge gates. The PL plane contains the Q1's of all the horizontal and T-shaped state gates and output gates. The V_Top, and V_Bottom planes contain the Q1's and Q2's of all the vertical output gates. There are also four clock buffer regions, TL, TR, BL and BR, which contain the clock buffers for clock signals Φ1 and Φ2. Apart from the three types of CLORA gates (i.e. edge gates, state gates and output gates), static CMOS inverter gates are also used in a CLORA FSM. These CMOS inverters are used to generate the complementary values of the FSM inputs and the true values of the FSM outputs. Figure 3.14 depicts how the inverter is laid out.

3.7.1 Gate Schematic

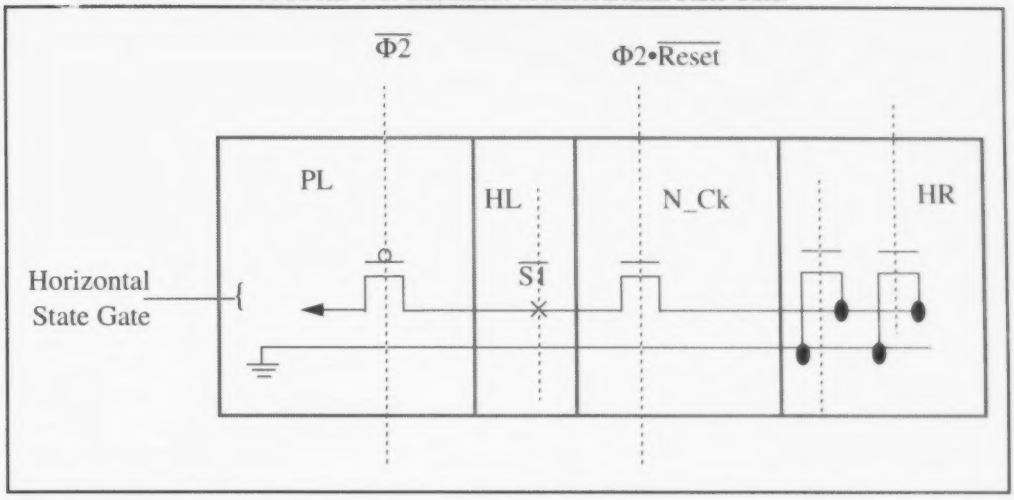
All the FSM inputs enter the CLORA array from the left edge of the PL plane via metal. Then, they are connected to vertical polysilicon strips and are used as inputs to the edge gates.

FIGURE 3.14 Schematic of a Edge Gate and a Static CMOS inverter.



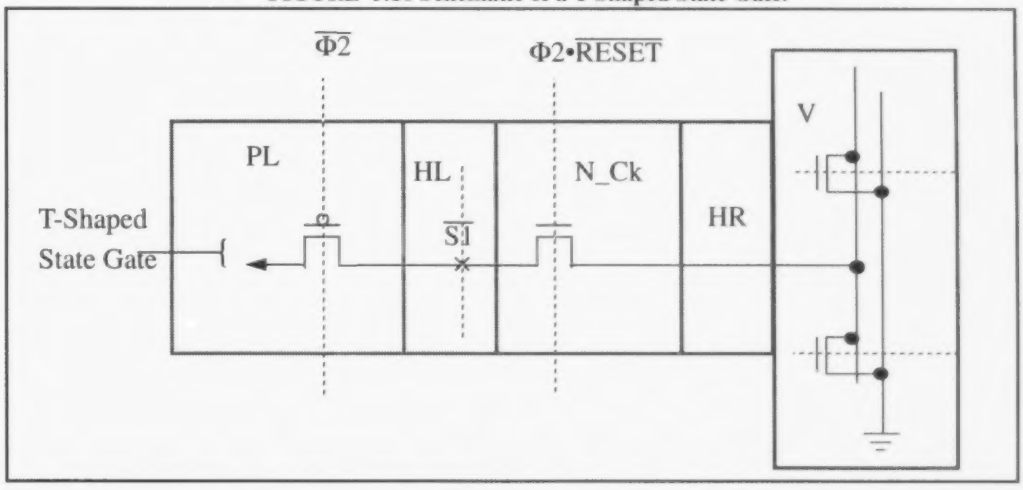
As shown in Figure 3.14, edge gates are laid out horizontally occupying an entire row of the array. The input sections of the edge gates are on the HL plane while the output sections of the edge gates lie on both the HR and the V planes. In the HR plane, the edge gate outputs are connected to vertical polysilicon strips which serve as inputs to the state gates. Within the V plane, the outputs of the edge gates are connected to horizontal polysilicon strips. These polysilicon strips are used as inputs to the vertical or T-shaped gates.

FIGURE 3.15 Schematic of a Horizontal State Gate.



The state gates can be laid out either horizontally or T-shaped. When a state gate is laid out horizontally (Figure 3.15), it is a mirror image of the edge gates. Its input section lies on the HR plane because it takes the outputs of the edge gates as inputs. Also, its output section lies on the HL plane because its output is taken by the edge gates as input. The output of a state gate is connected to a vertical polysilicon strip. This polysilicon strip is used as input to the edge gates.

FIGURE 3.16 Schematic of a T-Shaped State Gate.

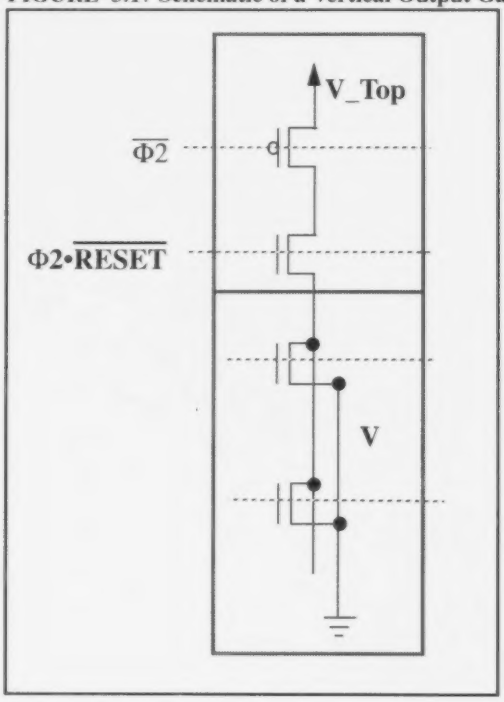


When a state gate is laid out T-shaped (Figure 3.16), its input section lies on the V plane; while its output section lies on the HL plane just as in the horizontal case. Laying out a

state gate this way allows us to solve the high fan-in state gate problem (to be explained later).

The output gates can be laid out horizontally, T-shaped or vertically. When an output gate is laid out horizontally or T-shaped, it has the same layout as a state gate. However, its output is no longer connected to a vertical polysilicon strip. Instead, its output is routed to exit the array from the PL or the PR planes (Figure 3.13).

FIGURE 3.17 Schematic of a Vertical Output Gate.



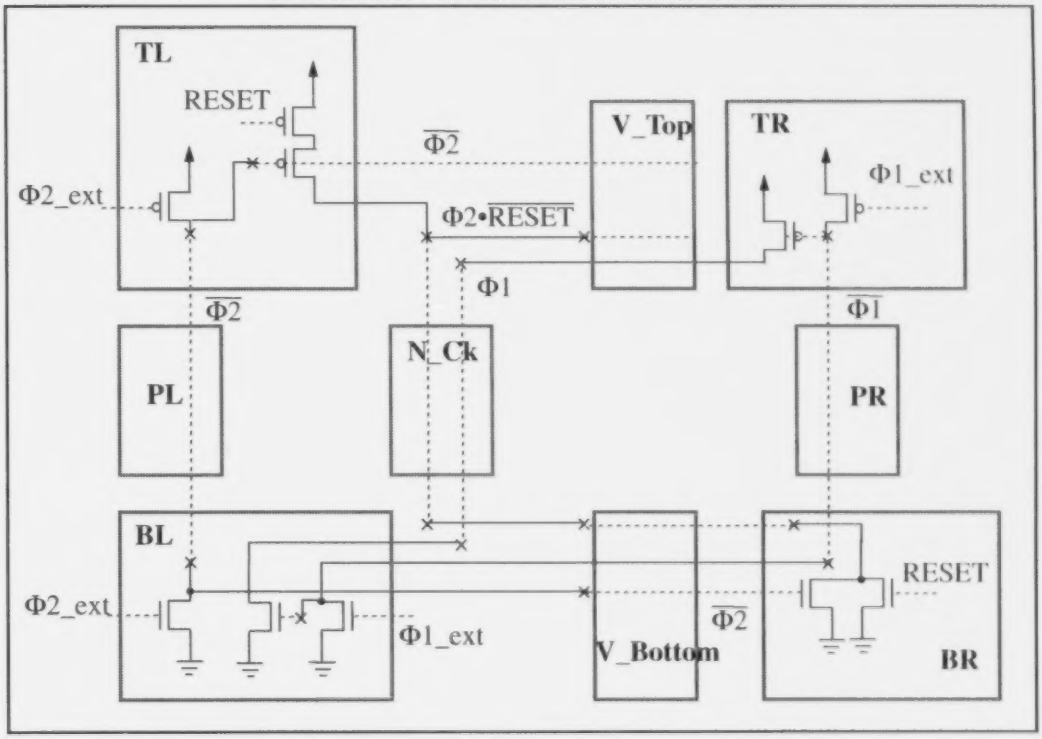
When an output gate is laid out vertically, its input section is placed in the V plane, and its output section can either lie on top of the V_plane (i.e. the V_Top plane) or at the bottom of the V plane (i.e. V_Bottom plane). As an example, a vertical output gate with its output section in the V_Top plane is shown in Figure 3.17.

3.7.2 Clock Buffers

In order to ensure the correct timing relationship between Φ and $\overline{\Phi}$, the way the clock signals are buffered and laid out in the FSM structure has to be designed very carefully. In

the following description, the clock signals that are fed to the CLORA FSM are labelled Φ1_ext, and Φ2_ext. Thus, the reader can easily distinguish between the internal clock signals (i.e. the actual clock signals used by Q1's and Q2's), Φ1, and Φ2 from the external clock signals, Φ1_ext, and Φ2_ext.

FIGURE 3.18 Schematic of the Clock Buffer Planes.



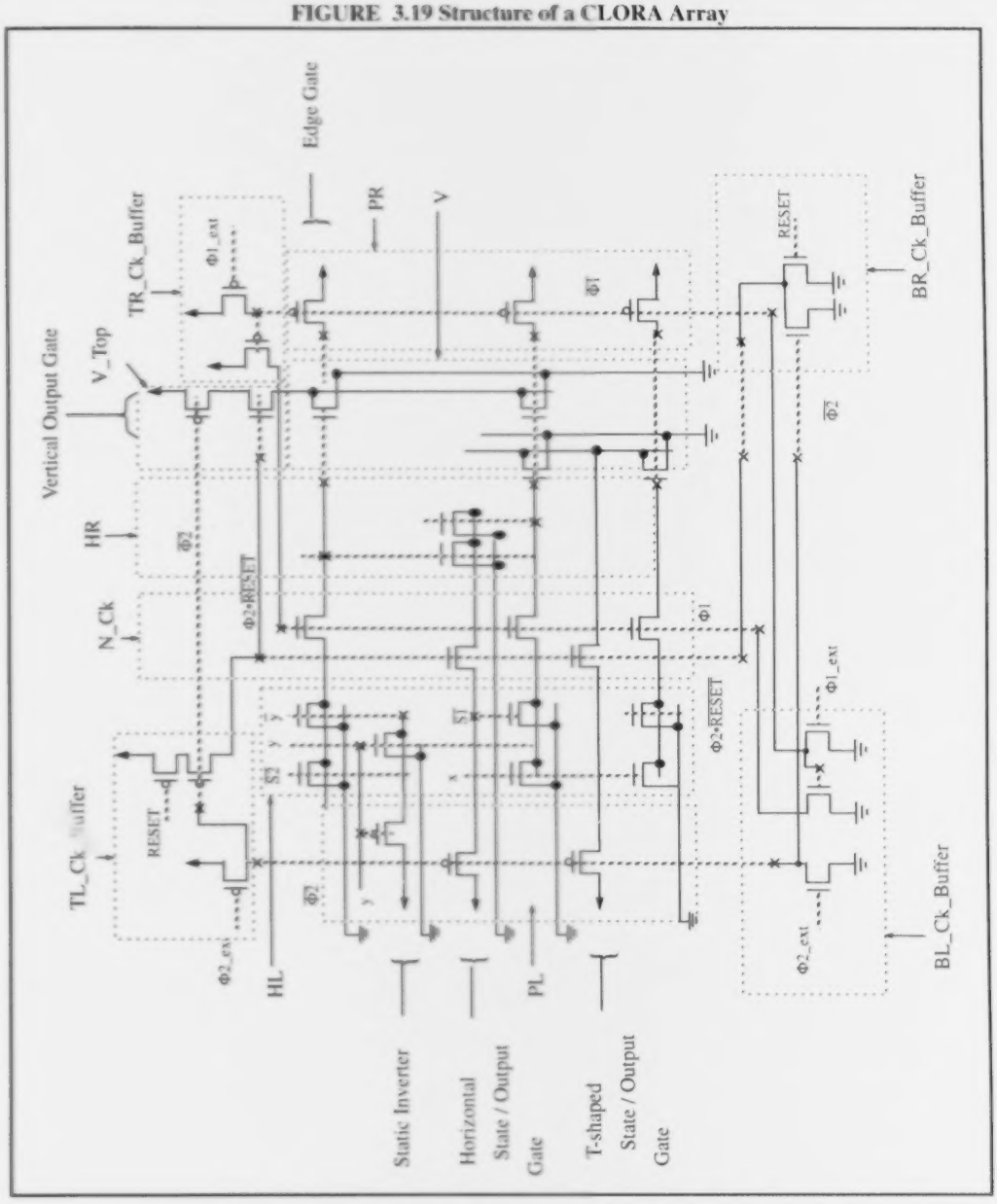
First, Φ1_ext is fed to a static CMOS inverter to generate $\overline{\Phi 1}$. Then, $\overline{\Phi 1}$ is fed into another CMOS inverter to produce Φ1. Then, Φ2_ext is fed into a CMOS inverter to produce $\overline{\Phi 2}$. Next, $\overline{\Phi 2}$ is NORed with the reset signal, RESET, using a CMOS NOR gate to produce $\Phi 2 \bullet \overline{RESET}$. In order to ensure the low-to-high switching of $\overline{\Phi 1}$ / $\overline{\Phi 2}$ (i.e. Q1 being turned off) leads the high-to-low switching of $\Phi 1$ / $\Phi 2 \bullet \overline{RESET}$ (i.e. Q2 being turned off), a special way shown in Figure 3.18 has been used to lay out the clock buffers. In Figure 3.18, all the P-type and N-type transistors of the clock buffer gates are laid out on the top and at the bottom of the CLORA array, respectively, with their output nodes running vertically down the array via polysilicon strips. When Φ1_ext switches from high to low, the resulting $\overline{\Phi 1}$ is made available through the P transistors. Thus, the rising edge on the

$\overline{\Phi 1}$ at the top of the polysilicon strip always leads the rising edge on the $\overline{\Phi 1}$ at the bottom of the polysilicon strip. When $\overline{\Phi 1}$ switches from low to high, the resulting $\Phi 1$ is made available through the N transistors of the next CMOS inverter. Thus, the falling edge on the $\Phi 1$ at the bottom of the polysilicon strip always leads the falling edge on the $\Phi 1$ at the top of the polysilicon strip. Therefore, the relationship that the rising edge of the complementary clock signal $\overline{\Phi 1}$ always leads the falling edge of the clock signal $\Phi 1$ is kept for all CLORA gates. Same theory applies to $\Phi 2\_ext$, $\Phi 2 \bullet \overline{RESET}$, and $\overline{\Phi 2}$. Thus, the proper timing relationship between $\Phi 2 \bullet \overline{RESET}$, and $\overline{\Phi 2}$ is kept.

3.7.3 Schematic of the CLORA Array

After describing the schematics of the various components in the CLORA array, a schematic of the entire CLORA array structure is shown in Figure 3.19.

FIGURE 3.19 Structure of a CLORA Array



Looking at Figure 3.19, the way this layout structure works is very straight forward. First, inputs coming in from the left hand side of the FSM are ready when Φ1 is high. Then, the edge gates are evaluated, and their outputs are fed into the state gates and the output gates. When Φ2 is high, the state gates and the output gates are evaluated, and their outputs are either fed into the edge gates or used by the data path.

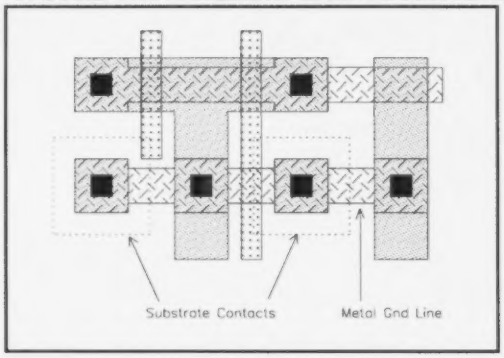
3.7.4 Physical Layout of the CLORA Array

After describing the schematic of the CLORA array, the way to actually lay out the CLORA gates using layers of polysilicon, N and P diffusion, and metal is described in the following sections.

3.7.4.1 HL, HR, and V Planes

Since both HL and HR planes contain the input and output sections of the horizontal CLORA NOR gates, they are laid out similarly. On each plane, there are strips of polysilicon and N-diffusion running vertically down the plane, and strips of metal running horizontally across the plane. The polysilicon lines serve as both the gates of transistors and interconnect for the input signals to the edge gates. Since these input signals are only used in a few places, multiple column folding (i.e. two or more polysilicon strips sharing the same column) can reduce the widths of these planes.

FIGURE 3.20 A Transistor in the CLORA Array.



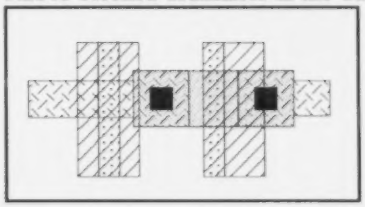
The N-diffusion strips serve as the source of the parallel transistors of the CLORA NOR gates. These transistors are grounded by connections to rows of grounded metal at regular intervals. As shown in Figure 3.20, to form a transistor, a tab of diffusion is extended out from the N-diffusion strip. With the vertical N-diffusion as the source and the other end of the diffusion as the drain, a transistor is formed by running vertical polysilicon through the middle as the gate. The horizontal metal is used to connect all the drains of the transistors to form a parallel structure. The V plane is similar to an HL plane except rotated 90

degrees counter-clockwise. It has polysilicon and N-diffusion running horizontally, and metal running vertically. However, the polysilicon strips are no longer folded.

3.7.4.2 PL, PR, V_Top, V_Bottom and N_Ck Planes

The PL plane contains the Q1's of all the state gates and horizontal output gates. It also contains the P-type transistors of all the static CMOS inverters. The PR plane contains the Q1's of all the edge gates. The N_Ck plane contains all the Q2's of the edge gates, the horizontal and T-shaped state gates and output gates. The transistors in the PR, PL, and the N_Ck planes are made up of horizontal tabs of N-diffusion with vertical polysilicon strips carrying the appropriate clock signals running through them (Figure 3.21). The V_Top, and V_Bottom planes contain both Q1's and Q2's of all the vertical output gates. The transistors in the V_Top, and the V_Bottom planes are made up of vertical tabs of N-diffusion with horizontal polysilicon strips carrying the appropriate clock signals running through them.

FIGURE 3.21 A Clocked Transistor in the CLORA Array.



3.7.5 Pros and Cons of the CLORA Array

The CLORA array structure can be viewed as a hybrid of the Weinberger array and the PLA. The advantages and disadvantages of the CLORA array are as follows:

Advantages:

1. The HR and HL planes are similar to a Weinberger array in that they can be multiple-column-folded to reduce the width of the array.
2. Multi-level logic functions can be implemented in the HR and HL planes.

3. The V plane is similar to the OR plane of the PLA in that it can also be multiple-column-folded to reduce the width of the array.

Disadvantages:

1. The size of the clock buffers must be increased as the size of the array increases.
2. Long clock lines.

3.8 Fully-Encoded FSM realization

This section presents how to lay out a fully-encoded FSM using the CLORA array. A fully-encoded FSM means that the states of the N-state FSM is encoded using $\log_2 N$ number of state variables. The purpose of this section is to show that CLORA logic can also lay out a fully-encoded FSM. Implementing a fully-encoded FSM using CLORA NOR gates is very similar to using a PLA implementation. First, the FSM has to go through a logic optimization stage which assigns a binary code to each of the FSM states, and then optimizes the two-level logic expressions of the FSM's combinational logic. Then, instead of using a PLA to implement the two-level logic description of the FSM, we use a CLORA array. One set of CLORA NOR gates controlled by $\Phi 1$ are used in evaluating the products (i.e. the AND plane of a PLA). This set of CLORA gates are laid out the same way as the edge gates in the un-encoded FSM. Another set of CLORA NOR gates controlled by $\Phi 2$ are used in evaluating the outputs and the state variables of the FSM (i.e. the OR plane of a PLA). This set of CLORA gates are laid out the same way as the state or output gates in the un-encoded FSM.

The main difference between the implementation of an un-encoded FSM and an encoded FSM is that there is no HR plane in the encoded FSM layout. We chose this approach because the state gates in a fully-encoded FSM usually has many product terms as inputs (i.e. high fan-in). If we lay them out horizontally, we will end up having a very wide HR plane.

3.9 Summary

A new type of logic gate, the CLORA gate, and the way to use it to implement an FSM have been presented in this chapter. CLORA logic is partially dynamic since the value of the output node is at times held as charge on the gate output. It is also partially ratioed logic because, when both Q1 and Q2 are on, the whole CLORA gate behaves like a ratioed logic gate. Both fully-encoded or un-encoded FSMs can be implemented and laid out in a CLORA array structure. In order to investigate how efficient our new layout method is compared with PLAs, this new method has to be applied to practical FSM examples. To do this, a CLORA FSM generator has been developed. In the next chapter, the development of this CLORA FSM is discussed in detail.

CHAPTER 4

Implementation

In order to automate the process of generating a CLORA FSM, a layout generator has been developed. As well, a folded PLA generator has also been developed so that we can compare our results with the results given by a folded PLA. Both generators are developed under the design environment of Cadence [15], which is a computer aided design system, using CMOS4S technology. The implementation details of these two layout generators are presented in this chapter.

4.1 Cadence

Before going into detail explaining the layout generators, some of the features of Cadence need to be introduced first so as to help the reader understand how the generators are implemented.

In Cadence, an integrated circuit is represented in different forms such as physical layout, schematic, symbol, etc.. Each of these different forms is called a *representation*. Cadence supports a programming language called IL. This language is based on the programming language LISP and has a C-like syntax. Apart from the basic IL language, two other languages, SKILL and dbAccess, that are based upon IL, are provided by Cadence. SKILL, which is an extension of IL, is a graphics interface language. dbAccess is a database access language.

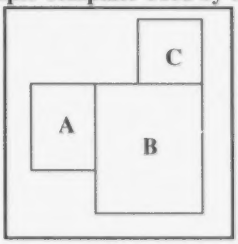
Apart from these programming languages, Cadence also provides several collections of function libraries such as Module Maker (MM) and MCIL. MM consists of functions which allows a user to build their own module generators. Using the MM functions, a developer can write an MM module generator program that builds circuits in the way specified by the users. An MM module generator program consists of different sections of code, called stages. Every MM module generator must have an init stage which is responsible for declaring and obtaining global information, such as the names of all the

inputs and outputs, about the circuit. Then additional stages which have access to the global information can be developed to fit the developer's intent. Except for the init stage, executing a stage generates a *representation*, which is the name of the stage, of the circuit. Thus, if a 'layout' stage of an MM generator is executed, a 'layout' *representation* of a circuit will be generated.

MCIL is a collection of SKILL-based functions that is specifically intended for use in conjunction with the Structure Compiler (to be explained later) in Cadence. MCIL allows the user to manipulate arrays generated by the Structure Compiler.

In both of our layout generators, we have two stages: init and layout. In the layout stage, we use the Structure Compiler in Cadence to generate the actual circuit layout. The Structure Compiler (SC) allows the user to lay out repetitive or semi-repetitive structures without having to place each cell individually. Using the SC, a layout is generated in two steps. First, a template, which is an abstract composition of the layout structure, is created using the set of *plane* commands in the SC menu. A sample template used by the SC is shown in Figure 4.1.

FIGURE 4.1 A Sample Template Used by the Structure Complier.



A template consists of one or more array blocks illustrated by A, B and C in Figure 4.1 above. Each array block has its own set of properties which are used to control how it is synthesized by the SC. Also, how the array blocks fit together in the final layout is indicated by the relative locations and the alignments among array blocks drawn in the template. For example, in Figure 4.1, array blocks A and B are aligned at the top edge and array blocks B and C are aligned at the right edge. The properties of the array blocks used in this project are as follows:

- Name:

  This is the name of the array block that appears in the template.

- Type:

  An array can either be one of two types: homogeneous or heterogeneous. A homogeneous array is one that contains only one master cell. Thus, all cells in a homogeneous array are identical, except perhaps for their orientation. A heterogeneous array is one that contains more than one master cell.

- Personality:

  A symbolic description of an heterogeneous array is an array of symbols where each symbol represents a single master cell. For each heterogeneous array, there is an associated symbolic description. A personality file of a template contains all the symbolic descriptions of all the heterogeneous array blocks within the template. To distinguish which symbolic description is for which heterogeneous array block, each symbolic description block in the personality file is preceded with a ".plane" command followed by the name of the symbolic description block.

  The personality property of an array block is the name of the symbolic description block appears in the personality file. Since only heterogeneous arrays require symbolic descriptions, this property is only relevant to heterogeneous arrays. A sample personality file is shown in Figure 4.2.

FIGURE 4.2 A Sample Personality File Used by the Structure Complier.

```
.plane AndTopDrives OrTopDrives
__<> H

.plane AndJoinTop
__|||

.plane And Or
sxslsl pp
lsllxs lp
llllssl pq
llllsss lq
lsssxll pp
lsxslx pp

.plane AndBottomDrives OrBottomDrives
>_>_ J

.plane AndJoinBottom
```

- Set:

This property is used to specify the number of columns and rows in a homogeneous array. This property is irrelevant to heterogeneous arrays because the size of the array is intrinsically specified in the personality file.

- masters:

  This property contains the name and *representation* of the cells used in generating the array. For a homogeneous array, only the name and the *representation* of the cell are needed. For each of the cells used in generating a heterogeneous array, both the name and the *representation* of the cell and the single character (i.e. symbol) in the personality file associated with this cell are needed.

- Orientation Pattern:

  This property specifies the orientations of the cells in the array block.

- Pins:

  This property is used to define pins in the array.

- Procedure:

  This property is used to specify a procedure to be executed after synthesizing the corresponding array. This procedure can make use of a set of SKILL-based array manipulation functions called MCIL.

After creating the template for the layout, the user can invoke the SC either by executing the "compile array structure" command in the SC menu, or by calling a SKILL function called compileArray(). Then, the SC will synthesize the layout according to the given template.

4.2 Overview of Programmable Logic Array Layout Generator

In determining how our new layout method, CLORA FSM, compares with other methods, FSMs laid out with PLAs are used as a reference. Unfortunately, only an unfolded PLA generator was available in the Module Maker package distributed to the University of

Alberta; therefore, we needed to implement a folded PLA generator to obtain a better comparison. The Module Maker package that is available in the Cadence distributed by the Canadian Microelectronics Corporation (CMC) was developed at the University of British Columbia [16]. This package contains several module generators such as RAM, ROM, FSM and PLA. In order to develop a folded PLA generator in the shortest amount of time possible, we decided to address the problem in the following way. First, using existing application programs that were available to us, we obtained a matrix representation of a folded PLA layout. Then, the SC in Cadence was used to generate the folded PLA layout. During the second stage, we will make use of some of the functions in the UBC's Module Maker package.

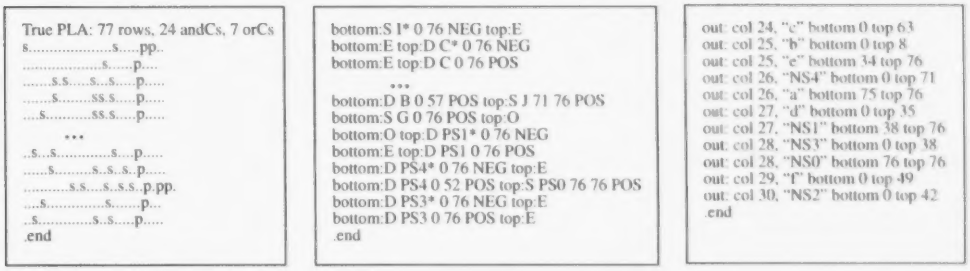
In order to obtain the matrix representation of the folded PLA, we decided to make use of an intermediate result generated by a PLA generator, *anchor*, in OCTTOOLS 5.1. OCTTOOLS is a collection of CAD programs and libraries developed at the University of California, Berkeley [17]. By operating on the same data base accessed via the OCT data manager, the tools form a system for integrated circuit design. The OCTTOOLS PLA generator, *anchor*, reads in a file describing an unfolded PLA and produces a folded or unfolded PLA layout in the specified technology. In order to access the intermediate result of *anchor*, modifications were made to the program so that it outputs the information required to lay out a folded PLA in a suitable format. Then, a procedure called Folded_PLA() is executed within Cadence. This procedure invokes the various stages of a folded PLA MM generator which reads in the information produced by *anchor*, and generates the folded PLA layout.

4.3 *Anchor*'s Output

Anchor takes a PLA description file as input and generates a PLA layout, simple column folded or unfolded, in a format that only the OCTTOOLS graphics editor, VEM can read. To make everything Cadence compatible, we decided to feed the intermediate results of *anchor* into Cadence to create a PLA layout in Cadence's format.

After slightly modifying *anchor*, it now generates three new information files. The first is the .andor file as shown in Figure 4.3(a). This file contains the personality matrix of the AND plane and the OR plane of the folded PLA. This matrix indicates the size of the array and the locations of the transistors in the AND plane and the OR plane. An 's' represents a transistor in the AND plane. A 'p' represents a transistor in the OR plane. A '.' represents interconnect.

FIGURE 4.3 a) A Sample .andor File; b) A Sample .input File; c) A Sample .output file.


True PLA: 77 rows, 24 andCs, 7 orCs
s.....s.....pp..
.....s.....p....
.....s.s.....s....s.....p.....
.....s.....ss.s.....p.....
....s.....ss.s.....p.....
...
..s...s.....s....p.....
.....s.....s..s..s..p.....
.....s.s....s..s.s..p.pp.
....s.....s.....p...
..s.....s..s.....p.....
.end
bottom:S 1* 0 76 NEG top:E
bottom:E top:D C* 0 76 NEG
bottom:E top:D C 0 76 POS
...
bottom:D B 0 57 POS top:S J 71 76 POS
bottom:S G 0 76 POS top:O
bottom:O top:D PS1* 0 76 NEG
bottom:E top:D PS1 0 76 POS
bottom:D PS4* 0 76 NEG top:E
bottom:D PS4 0 52 POS top:S PS0 76 76 POS
bottom:D PS3* 0 76 NEG top:E
bottom:D PS3 0 76 POS top:E
.end
out: col 24, "c" bottom 0 top 63
out: col 25, "b" bottom 0 top 8
out: col 25, "e" bottom 34 top 76
out: col 26, "NS4" bottom 0 top 71
out: col 26, "a" bottom 75 top 76
out: col 27, "d" bottom 0 top 35
out: col 27, "NS1" bottom 38 top 76
out: col 28, "NS3" bottom 0 top 38
out: col 28, "NS0" bottom 76 top 76
out: col 29, "f" bottom 0 top 49
out: col 30, "NS2" bottom 0 top 42
.end

(a) (b) (c)

The second information file is the .input file. An example is shown in Figure 4.3(b). This file contains the names of the input signals and the locations where they enter the PLA. Each row in the file corresponds to an AND plane column in the PLA. Since, for a folded PLA, inputs can enter the array from either the bottom or the top of the array, each row in the file contains a bottom section and a top section. The symbols used to describe the two sections are as follows.

- 'E':

  There is no signal going into the AND plane from this location.

- 'D':

  There is a signal going into the AND plane from this location. The name, row range, and the phase (i.e. NEG or POS) of the signal are given following the 'D' symbol. Moreover, this also implies that the complementary signal will enter the array through either the column that is to the left of or right of this column.

- 'S':

Same as 'D', except that the complement of the signal is not used in the array. Since the signal buffer is two columns wide, a column space has to be found in the array or added to the array when the data is processed by Cadence.

- 'O':

  There is no signal going into the AND plane from this section of the column. However, this section is reserved for the signal buffer of the signal that runs to the left of this column in the same section. This means that if an 'O' appears in either the top or the bottom section of a column, an 'S' MUST be present in the same section to the left of that column.

The third file generated by the modified *anchor* is the .output file, as shown in Figure 4.3(c). This file contains the names of the output signals, their locations, and their row ranges in the OR plane of the folded PLA. The range of each signal contains the lower range (i.e. bottom row) and the upper range (i.e. top row) of the signal. Also, for Cadence to tell whether an output signal exits from the top or bottom or the OR plane, the lower range of a signal must be equal to 0, or the upper range of a signal must be equal to the number of rows in the array less one.

Having the three information files, Cadence will be able to generate the simple column folded PLA.

4.4 Folded PLA MM Generator

After obtaining the three new information files from *anchor*, the folded PLA can be generated by running the IL procedure Folded_PLA(). This procedure will set the technology environment, and execute the init stage and the layout stage of the folded PLA MM generator to generate the layout *representation* of the folded PLA. Both stages of the MM generator will be described in detail in the following sections.

4.4.1 The Init Stage

The init stage of the folded PLA MM generator is used to declare all the global parameters and terminals. It also executes a procedure which will read the files generated by *anchor* into Cadence. In the init stage, an MM function, defParamters(), is used to prompt the user for parameters used in the folded PLA generation. The user-settable parameters are as follows:

1. Block Name:

   This is the name of the folded PLA the user is generating.

2. Template directory:

   This is the location of the template directory used by the Structure Complier.

3. Specific template:

   The *representation* of the template to be used to construct the folded PLA layout. A default folded PLA template called andor is created for the folded PLA layout.

4. Cell directory:

   This is the location of the three input files generated by the modified *anchor* (i.e. .andor, .input, and .output).

5. Cell name:

   Name of the input cell. The generator uses this name to form the names of the files generated by *anchor*.

6. Spacing of substrate contact rows (in number of rows):

   The spacing between substrate contact rows in the AND plane. By default, it is 2.

7. Spacing of substrate contact columns (in number of columns):

   The spacing between substrate contact columns in the OR plane. By default, it is 2.

After obtaining all the necessary information, the generator initializes all the global variables. Then a procedure called OCTRead is executed to process the information provided by the three *anchor* files. Using the files, the input and output terminals which

are global to all *representation*s of the folded PLA are declared by using the MM function defTerminals().

4.4.2 OCTRead

OCTRead is an IL procedure used to read in all the data provided by the .andor, .input, and .output files into Cadence. It is also responsible for re-processing the personality matrix generated by *anchor* into a form that can be used by the SC.

First, OCTRead initializes all the global variables used in the program. Then, it starts to read in the personality matrix of the AND plane and the OR plane (i.e. the PLA core) in the .andor file, the input driver information in the .input file, and the output driver information in the .output file. After that, it starts to process the personality matrix of the PLA core while generating the personality matrices of other array blocks in the PLA template. First, the vertical signals of both the AND plane and the OR plane have to be broken into two sections since there are two distinct sections, top and bottom, in each column. This can be done by making use of the signal range information provided in the .input and the .output files. The personality matrix of the PLA core is then updated to reflect the changes. Now, the personality matrix of the OR plane is ready to be used by the SC. Next, a matrix which contains the symbols of substrate rows are created according to the substrate spacing given by the user. The next thing to do is to generate personality matrices for other array blocks in the PLA template, such as the AND driver array blocks, and the OR driver array blocks. To generate the personality matrices for the AND driver array blocks, the information given in the .input file is used. According to the status of the top and the bottom sections of all the columns, the AND driver personality matrices are built. During this process, extra columns may be added to the personality matrix of the AND plane in order to fit the input driver in. After this, the personality matrix of the AND plane is ready to be used by the SC. The generation of the matrices of the OR driver array blocks are straight forward. If there is an output exiting from a location, the symbol that indicates the presence of an output driver will be written into the corresponding matrix position. Finally, all the personality matrices for the different array blocks present in the

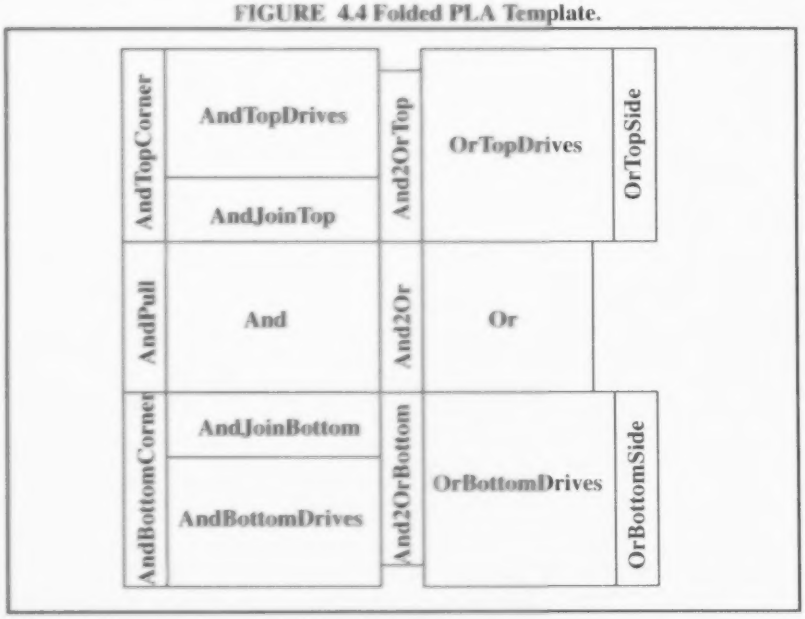
template will be written to an output file with a .m extension. This .m file is ready to be used as the personality file for the SC.

4.4.3 The Layout Stage

After executing the init stage, the Folded_PLA() procedure will execute the layout stage of the folded PLA MM generator. In the layout stage, the MM generator executes a procedure called Folded_PLALayout(). This procedure uses the SC by calling the function compileArray(). Using the pre-defined folded PLA template, a set of pre-defined master cells and the .m file as the personality file, the SC generates a pseudo-NMOS, NOR-NOR structured folded PLA layout in CMOS4S technology.

4.4.4 Folded PLA template

FIGURE 4.4 Folded PLA Template.



The default folded PLA template is shown in Figure 4.4. The template consists of sixteen array blocks or planes, and each of these array blocks is explained as follows:

1. And plane (Heterogenous):

This plane makes up the entire AND plane of the folded PLA. A procedure called Folded_andAddRows() is specified in the procedure field. This procedure is responsible for adding the substrate contact rows into the AND plane.

2. Or plane (Heterogenous):

   This plane makes up the entire OR plane of the folded PLA. A procedure called Folded_orAddSubContacts() is specified in the procedure field. This procedure is responsible for adding the substrate contact columns, and the extra row space taken up by the AND plane's substrate contact rows to the OR plane.

3. AndTopDrives, AndBottomDrives (Heterogenous):

   These two planes make up the input driver portions of the folded PLA. Procedures Folded_andTopDrivePromotePins() and Folded_andBottomDrivePromotePins() are used by these two planes, respectively. These two procedures are responsible for promoting the pins in the master cells to the final layout.

4. AndJoinTop, AndJoinBottom (Heterogenous):

   These two planes consist of the interfacing circuitry between the AND plane and the input driver planes.

5. OrBottomDrives,OrTopDrives (Heterogenous):

   These two planes make up the output driver portions of the folded PLA. Procedures Folded_orTopDriveProcess() and Folded_orBottomDriveProcess() are used by these two planes, respectively. These two procedures are responsible for promoting the pins in the master cells to the final layout, and adding the extra column space taken up by the OR plane's substrate contact columns to the corresponding driver plane.

6. AndPull plane (Homogenous):

   This plane consists of the pull-up transistors of the NOR gates in the AND plane. A procedure called Folded_andPullAddRows() is specified in the procedure field. This procedure is responsible for adding the extra row space taken up by the AND plane's substrate contact rows to the AndPull plane.

7. And2Or plane (Homogenous):

   This plane consists of the interfacing circuitry between the AND plane and the OR plane. A procedure called Folded_and2orAddRows() is specified in the procedure field. This procedure is responsible for adding the extra row space taken up by the AND plane's substrate contact rows to the and2or plane.

8. And2OrTop, And2OrBottom, AndTopCorner, AndBottomCorner, OrTopSide, and OrBottom:

   Each of these planes contains a single cell which serves as interconnect between blocks.

4.4.5 Master Cells

As mentioned in Section 4.1, each array block is generated by placing the master cell(s) in the way specified by the orientation property, and by either the personality matrix (heterogenous array) or the set property (homogenous array). For the folded PLA, master cells have been created in CMOS4S technology. Thus, the SC can generate a folded PLA in CMOS4S technology.

4.5 Overview of the CLORA FSM Layout Generator

The second layout generator developed in this project is the CLORA FSM layout generator. Since this generator is important in testing our new FSM layout method, a lot of time has been spent in developing it. This generator basically consists of two main components. The first component is a C++ program called *clora* which generates a personality file that the SC can use as an FSM description file. The second component is a collection of IL, and MM programs which are used to generate the CLORA FSM layout. These IL and MM programs are executed by running a IL program, CLORA(), within Cadence.

FIGURE 4.5 Data Flow in the CLORA FSM Layout Generator.

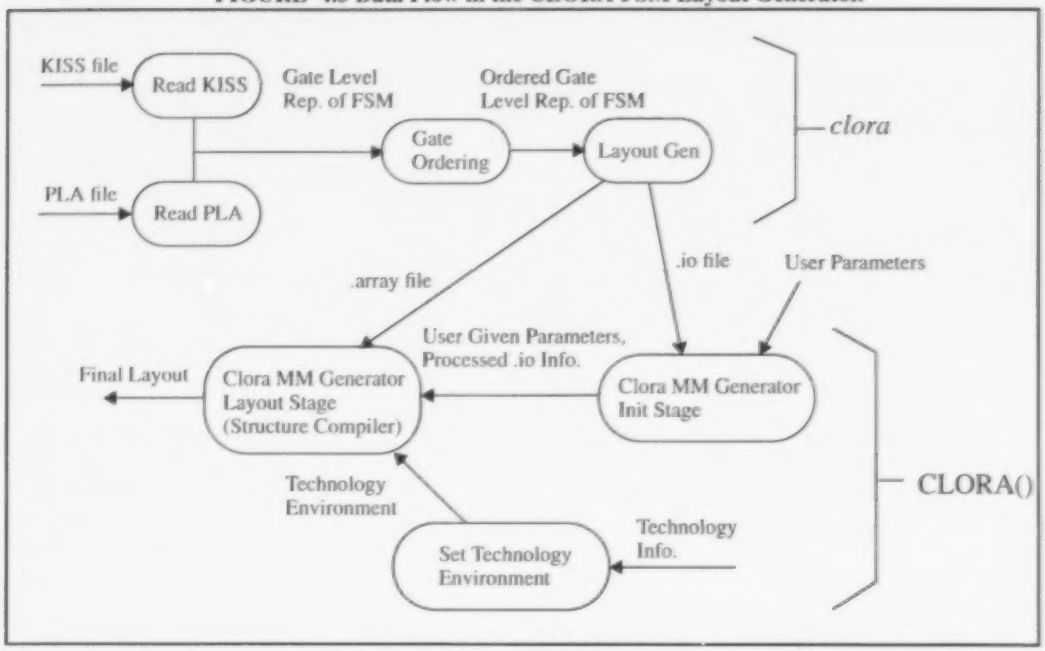


Figure 4.5 shows the data flow going from *clora* to CLORA(). There are three stages to the C++ program, *clora*. The first stage is Read KISS or Read PLA. This stage is responsible for converting the FSM description into a gate representation understood by the second stage. The second stage is called Gate Ordering and is responsible for ordering all the horizontal CLORA gates so that the widths of the HL and HR planes are minimized. The third stage, called Layout Gen, is responsible for placing the vertical signals of the HL and HR planes, the gate input sections in the V plane, and generating all the output files (i.e. .array, and .io).

Once the .array and .io files are available, an IL procedure called CLORA() is executed in Cadence. This procedure will then start initializing the layout environment and starts the init stage of a generator called the CLORA MM generator. The init stage of the CLORA MM generator will then initialize all the global variables. Then, CLORA() will execute the layout stage of the CLORA MM generator which calls the SC to generate the CLORA FSM layout using a CLORA array template.

In the following sections, detailed descriptions of all these components will be presented.

4.6 *clora*

clora is a C++ program which is run under the UNIX environment. It takes two kinds of inputs. The first kind is an FSM description in KISS format as shown in Figure 4.6. This input format is used when implementing an un-encoded FSM. The .s field specifies the number of states, the .p field specifies the number of products, and the .i and the .o fields specify the number of inputs and the number of outputs, respectively. The .ilb and the .ob fields specify the names of the inputs and the outputs. Each line in the main body of the data file consists of four fields: inputs, present state, next state, and outputs. Each of these fields is separated by a space.

The second input format accepted by *clora* is a circuit description in PLA format, as examplified in Figure 4.7. This format is used when implementing a fully-encoded FSM using the CLORA array.

The output generated by *clora* consists of two files. The first file is called .array which contains the personality matrices of all array blocks in the CLORA array template. This file is ready to be used by the SC in Cadence. The second file is called .io which contains the labels and the positions of all the inputs and outputs of the FSM.

4.6.1 Read KISS

FIGURE 4.6 Sample FSM description in KISS Format.

```
.i 2
.o 1
.p 11
.s 4
.ilb in1 in2
.ob out
-0 st0 st0 0
11 st0 st0 0
01 st0 st1 -
0- st1 st1 1
11 st1 st0 0
10 st1 st2 1
1- st2 st2 1
00 st2 st1 1
01 st2 st3 1
0- st3 st3 1
11 st3 st2 1
```

Read KISS is responsible for reading in an FSM description file in KISS format and forms the data structures used later in the program. Read KISS reads in the number of states, the number of products (edges), and the number of inputs and the number of outputs that the

FSM has. Then, it reads in the names of the inputs and the outputs. Next, it reads in the data from the input file line by line. For each line, it forms the signal nets for each of the inputs and their complements according to the input field, if they do not already exist. It then forms an edge gate and a signal net for the edge gate output using the information given in the input field and the present state field. If a state gate for the next state variable (i.e. next state gate) has not been created, Read KISS also forms a state gate and a signal net for the next state variable using the information given by the next state field. Then, the output of the edge gate will be added to the input list of the next state gate. This is because the edge is an entrance condition to the next state. For example, processing the third line of the main data body of the KISS file in Figure 4.6 gives us an edge gate with in1, $\overline{in2}$, and $\overline{st0}$ as inputs, and a state gate $\overline{st1}$.

The edge gate is stored under a data structure called H_Config. This implies that the edge gate is laid out horizontally. The state gate is placed either in a T-shaped fashion or horizontally depending on the number of inputs it has. The information given by the output field is stored for later use. After reading in the entire FSM description file, Read KISS starts to process the information about the state gates and the outputs.

To determine whether a state gate is laid out horizontally, or T-shaped, a profile describing the distribution of the number of inputs of the state gates is created. Then, using this profile, the decision is made according to the following pseudo-code:

```
int St_Pro[Max_in-degree of state gates];
/* St_Pro[i] = number of state gates which have i inputs.*/

Min_HR_Width = Min_in-degree of state gates;
for( i = Min_in-degree of state gates; i <= Max_in-degree of state gates; i++)
{
        if( i > Min_HR_Width && St_Pro[i] > (i - Min_HR_Width))
                Min_HR_Width = i;
}

for( j = 0; j <= number of state gates; j++)
{
        if( state[j]'s num_of_inputs <= Min_HR_Width)
                Lay out state[j] horizontally;
        else    Lay out state[j] in a T-shaped fashion;
}
```

In the above pseudo-code, the first 'if' statement is used to check whether it is worthwhile to layout state gates with i inputs horizontally. Assuming each T-shaped state gate occupies one column (i.e. no column sharing; the worst case), the number of extra columns required to lay out state gates that have i inputs in a T-shaped fashion is equal to St_Pro[i]. On the other hand, the number of extra columns required to lay out state gates that have i inputs horizontally is equal to the difference of i and the current HR width. Thus, if i is larger than the current HR width, the way to lay out the state gates is determined by comparing St_Pro[i] with i - Min_HR_Width.

State gates that are laid out horizontally are stored in the H_Config. Otherwise, they are stored in both H_Config and V_Config. This is because gates that are stored in V_Config will have their input sections laid out in the V plane. Thus, if a gate exists in both H_Config and V_Config, that gate will be laid out in a T-shaped fashion.

Knowing the number of 0-edges and the number of 1-edges of an output, Read KISS determines whether that output is laid out horizontally, in a T-shaped fashion with or without using an inverter, or vertically without using an inverter. The following pseudo-code describes how Read KISS makes the decision:

```
Min_HR_Width = the number of inputs of the highest fan-in state gate that is laid
out horizontally.

if (num_of_0-edges > Min_HR_Width && Min_HR_Width ≥ num_of_1-edges )
{
        form output gate using the 1-edges as inputs.
        the output gate is laid out horizontally.
        add extra inverter to invert the output.
}
else
{
        form output gate using the 0-edges as inputs.
        if (Min_HR_Width ≥ num_of_0-edges)
                the output gate is laid out horizontally.
        else
                the output gate is laid out vertically or in a T-shaped fashion.
                /* will be determined in a later stage */
}
```

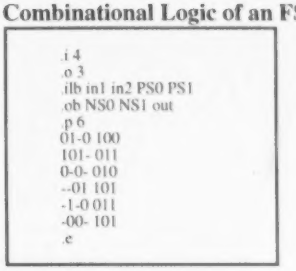
Output gates that are laid out horizontally are stored in the H_Config. Output gates that are laid out vertically are stored in V_Config. Output gates that are laid out in a T-shaped fashion are stored in both the H_Config and the V_Config.

After forming the gate representation of the FSM, the FSM is ready to be laid out. However, to minimize the area of the FSM, all the horizontal gates in the H_Config have to be ordered such that all the signal nets can be placed using the minimum number of vertical tracks possible. The ordering of the H_config is performed by the Gate Ordering stage, and will be described later.

4.6.2 Read PLA

Read PLA is similar to Read KISS. Instead of reading in the FSM description file, it reads in a circuit description file in PLA format (i.e. a .pla file) and forms the data structures used later on in the program. This section of code is used in place of READ FSM in implementing a fully-encoded FSM. A sample two-level logic expressions of an FSM in the PLA format (i.e. a .pla file) is shown in Figure 4.7. The first part of the .pla file is a set of control information. The .s, .i, .o, and .p fields specify the number of state variables, inputs, outputs, and product terms, respectively, of the PLA. The .ilb, and .ob fields specify the inputs and output labels respectively. The second part of the .pla file contains a symbolic description of the AND plane (the plane on the left), and the OR plane (the plane on the right) in which each line represents a product term. A '1' in the AND plane means that there is a connection between the product term and the true value of the corresponding input. A '0' means that there is a connection between the product term and the complementary value of the input. A '-' means that there is no connection. A '1' in the OR plane means that there is a connection between the product term and the output. A '0' means that there is no connection.

FIGURE 4.7 The Combinational Logic of an FSM in PLA Format.



Read PLA reads in the number of products, the number of inputs and the number of outputs that the PLA has. Then, it reads in the names of the inputs and the outputs. Next, it reads in the data from the input file line by line. For each line, it forms the signal net for each input and its complement according to the input field, if they do not already exist. It also forms a product gate and a signal net for the product using the information given in the AND field. Then, it stores the product gate in H_Config. The information given by the OR field is used to create the state and the output gates. If there is a '1' in a state variable or an output column, the product term corresponding to that line is added to the input list of the corresponding state or output gate. Then, static inverters are added to obtain the true values of the state variables and outputs. In a fully-encoded FSM, all the state gates and output gates are laid out in a T-shaped fashion.

After forming the gate representation of the fully-encoded FSM, the FSM is ready to be laid out. Again, in order to minimize the area of the FSM, all the product gates in the H_Config have to be ordered such that all the signal nets can be placed using the minimum number of vertical tracks possible. The ordering of the H_config is performed by the Gate Ordering stage, and will be described in the next section.

4.6.3 Gate Ordering

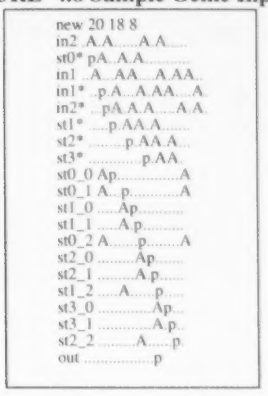
To order the gates in the H_Config, we uses an array optimizer in OCTTOOLS 5.1 called Genie [18]. Given a symbolic or a gate description of an array and a technology file, Genie orders and folds the nets in the array according to the user's specification. Before using Genie to optimize the HL and HR planes of the CLORA array, we have to generate a file that Genie can use as input. A function called Write_Genie_In() was written for this

purpose. To make full use of Genie's power, we have to determine how Genie is to be used and which input format we should give Genie. To do this, we listed the functions which we want Genie to accomplish.

1. Genie has to reduce the widths of both the HL plane, the HR plane and the V plane by ordering all the horizontal gates in H_Config and performing multiple column folding on their nets.
2. While performing column folding, Genie cannot mix the HL's signals, the HR's signals, and the gate input sections in the V plane together because they will be placed in different planes.

In order to satisfy the second restriction, we decided to use the -s option in Genie. The -s option is originally implemented for static CMOS circuits in which the P and the N parts must remain separate. However, since we have three separate planes instead of two, we have to determine which plane has to be separated from the other two. After performing some experiments, we have decided to separate the gate input sections in the V plane from the signal nets in the HR and the HL plane. To use the -s option to lay out the CLORA array, we create an input file with all the signals in the HL plane and the HR plane in the first half, and the gate input sections in the V plane (i.e. treating them as if they were signal nets) in the second half. Genie will treat the two halves as the P and the N parts of a static CMOS circuit. Thus, Genie will optimize the total width of the CLORA array without mixing up signals from different planes. However, in order to create such a file, the orientation between the input file and the actual CLORA array will be different (i.e. a row in the input file represents a column in the CLORA array). Thus, we will use the -r option (i.e. row folding), instead of the -c option (i.e. column folding). As an example, an input file used by Genie is shown in Figure 4.8. In this file, an 'A' represents a transistor, and a 'p' represents a contact.

FIGURE 4.8 Sample Genie Input File.



With the input file to Genie ready, Genie is run with the -r and the -s options within *clora*. After Genie is done, Genie stores its result in a file called "final". Then, a function in *clora* called Read_Final() will read in the result and order the gates stored in H_Config accordingly. Although Genie places the signals while it is doing the multiple row folding, this result is of no use to us. This is because the way that we are placing our signals is restricted by the following:

- In order to reduce the area of the FSM, the row (gate) pitch of the CLORA array is minimized. Thus, the layout does not allow two different vertical signals to be placed on the same column in adjacent rows.

Thus, the signals in the HL, the HR planes, and the gate input sections in the V plane have to be placed by a separate function in the Layout Gen stage in *clora*.

4.6.4 Layout Gen

This is the last stage in *clora*, and it involves placing the vertical signals in the HL and the HR planes, placing the gate input sections in the V plane, and generating the personality matrix file to be used by the SC in Cadence. To place the signals and the gate input sections, we employed the left-edge algorithm. The Left-edge algorithm is a very straight forward channel routing algorithm. The algorithm is described in the following pseudo-code:

```
Sig = array of unplaced signals;
```

```
Find out the 'hi' and 'lo' (i.e. the top and the bottom) of all the signals in Sig;
Sort the signals in Sig such that the 'lo' index is in an ascending order;

for( i = 0; i < num_of_signals; i++)
{
        if( Sig[i] can be placed in the available tracks)
                place Sig[i];
        else
                create new track and place Sig[i];
}
```

In *clora*, Quick Sort [19] has been used to sort the signals in order. Also, to solve the signal placing restriction posted in Section 4.6.3, the net length of each signal is extended by one row. This ensures that there is an empty space following every signal.

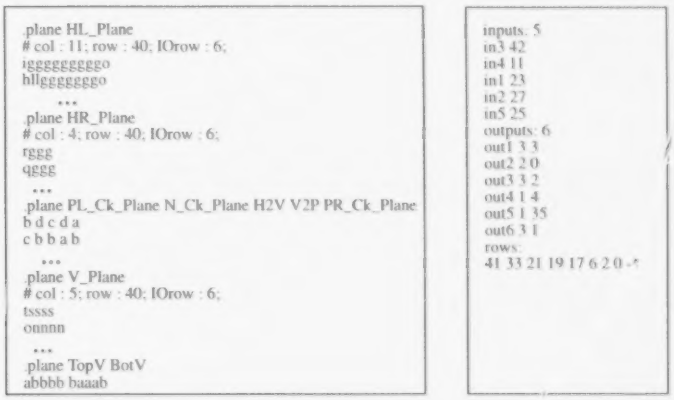
After placing all the signals in the HR and the HL planes, and all the gate input sections in the V plane, Layout Gen starts to determine whether an output gate should be placed vertically or T-shaped. The following pseudo-code explains how this decision is made.

```
V_Tracks = array of lists of V gates;
/* V_Track[i] contains list of V gates that have their input sections placed in
column i of the V plane; */
for( i = 0; i < num_of_V_tracks; i++)
{
        if( The first gate-input-section in V_Track[i] belongs to an non-inverted
        output gate (i.e. output gate formed by NORing the 0-edges) )
                place the output gate vertically with its Q1 and Q2 in the V_Top
                plane;
        else if( The last gate-input-section in V_Track[i] belongs to an
        non-inverted output gate)
                place the output gate vertically with its Q1 and Q2 in the V_Bottom
                plane;
        else    place the corresponding gates in V_Track[i] in a T-shaped fashion;
}
```

The output of an non-inverted output gate is ready to be used by the data path. Therefore, if the output gate is an non-inverting output gate and its input section is placed as the first or the last input section of a column, its output can exit the array directly through the top or the bottom. Hence, it is placed vertically. Otherwise, if the output gate is not an non-inverting output gate or its input section is not the first or the last input section of a column, it is placed in a T-shaped fashion.

Next, Layout Gen starts generating the matrix representations for all the heterogeneous array blocks in the CLORA array template. This process is straight forward because the locations of the gates and the signals of all the planes are well defined. Then, Layout Gen writes the matrix information into a file with a .array extension. Layout Gen also generates a file with a .io extension which contains the information about the inputs and the outputs of the FSM, such as their names, where they enter the CLORA array, and where they exit the CLORA array. This information is useful when Cadence assigns I/O terminals to the CLORA array layout. As an example, sample .array and .io files are shown in Figure 4.9.

FIGURE 4.9 a) Sample .array File; b) Sample .io File Generated by *clora*.



(a) (b)

4.7 CLORA MM Generator

Having the two information files from *clora*, the CLORA array can be generated by running the CLORA() procedure. This procedure will then execute the two stages of the CLORA MM generator, the init stage and the layout stage, to generate the CLORA array layout.

4.7.1 The Init Stage

The init stage of the CLORA array MM generator is used to declare all the global variables and terminals. It also, executes a procedure which will read into Cadence the output files generated by *clora*. In the init stage, defParamters() is again used to prompt

the user for parameters used in the CLORA array generation. The user-settable parameters are similar to that of the Folded_PLA MM generator, and are as follows:

1. Block Name:

   This is the name of the CLORA array the user is generating.

2. Template directory:

   This is the location of the template directory used by the Structure Complier.

3. Specific template:

   The *representation* of the template to be used to construct the CLORA array layout. A default CLORA array template called CLORA is created for the CLORA array layout.

4. Clora directory:

   This is the location of the two input files generated by *clora* (i.e. .array, and .io).

5. Clora name:

   Name of the input cell. The generator uses this name to form the names of the input files generated by *clora*.

6. Spacing of substrate contacts rows (in number of rows):

   The spacing between substrate contact rows in the HL and HR planes. By default, this is 6.

7. Spacing of substrate contacts columns (in number of columns):

   The spacing between substrate contact columns in the V plane. By default, this is 6.

After obtaining all the necessary information, the generator initializes all the global variables. Then a procedure called ReadFile() is executed to process the information provided by the .io file generated by *clora*. Next, using the information obtained by ReadFile(), the input and output terminals which are global to all *representation*s of the CLORA array are declared by using defTerminals().

4.7.2 ReadFile

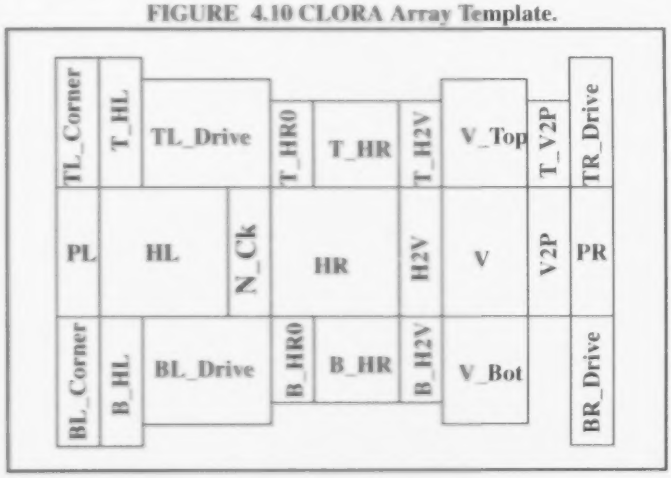
ReadFile is an IL procedure used to read in the data provided by the .io file into Cadence. First, it forms a list of input labels, and re-orders them according to the row via which they enter the array. Ordering the input labels helps Cadence to promote the input pins later on. Next, ReadFile forms a list of output labels for each of the four places that an output can exit the CLORA array (i.e. the top and bottom of the V plane, the PL plane and the PR plane). Then, it re-orders each list according the locations of the outputs. Again, ordering the output labels helps Cadence to promote the output pins later on.

4.7.3 The Layout Stage

After executing the init stage, the CLORA() procedure will execute the layout stage of the CLORA MM generator. In the layout stage, the MM generator executes a procedure called Clora_Layout(). This procedure uses the SC by calling the function compileArray(). Using the pre-defined CLORA template, a set of pre-defined master cells and the .array file as the personality file, the SC generates a CLORA array layout in CMOS4S technology.

4.7.4 CLORA Array template

FIGURE 4.10 CLORA Array Template.



The default CLORA array template is shown in Figure 4.4. The template consists of twenty-five array blocks or planes, and each of these array blocks is explained as follows:

1. HR plane, HL plane (Heterogenous):

   These two planes make up the HR and the HL planes of the CLORA array. Procedures called HR_Plane_AddSubRows() and HL_Plane_AddSubRows() are used by these two planes, respectively. These two procedures are responsible for adding the substrate contact rows to the planes.

2. N_Ck plane (Heterogenous):

   This plane makes up the N_Ck plane of the CLORA array. A procedure called N_Ck_Plane_AddSubRows() is specified in the procedure field. This procedure is responsible for adding the substrate contact rows into the N_Ck plane.

3. V2P plane (Heterogenous):

   This plane consists of the interfacing circuitry between the V plane and the PR plane of the CLORA array. A procedure called V2PAddSubRows() is specified in the procedure field. This procedure is responsible for adding the substrate contact rows into the V2P plane.

4. H2V plane (Heterogenous):

   This plane consists of the interfacing circuitry between the HR plane and the V plane of the CLORA array. A procedure called H2VAddSubRows() is specified in the procedure field. This procedure is responsible for adding the substrate contact rows into the H2V plane.

5. V plane (Heterogenous):

   This plane makes up the V plane of the CLORA array. A procedure called V_Plane_Process() is specified in the procedure field. This procedure is responsible for changing the orientation of the master cells in the V plane, adding the substrate contact columns, and the extra row space taken up by the HR plane's substrate contact rows to the V plane.

6. PL plane, PR plane (Heterogenous):

These two planes make up the PL and the PR planes of the CLORA array. Procedures PL_Plane_Process() and PR_Plane_Process() are used by these two planes, respectively. These two procedures are responsible for adding the substrate contact rows to the corresponding plane, and for promoting the I/O pins in the master cells to the final layout.

7. V_Top plane, V_Bottom plane (Heterogenous):

   These two planes make up the Q1's and Q2's of the vertical output gates of the CLORA array. Procedures V_Top_Process() and V_Bottom_Process() are used by these two planes, respectively. These two procedures are responsible for promoting the output pins in the master cells to the final layout, and for adding the extra column space taken up by the V plane's substrate contact columns to the corresponding plane.

8. TL_Drive, BL_Drive, TR_Drive, and BR_Drive planes:

   Each of these planes contains a single cell which is the circuitry of one of the four clock buffer regions.

9. TL_Corner, T_HL, T_HR0, T_HR, T_H2V, T_V2P, BL_Corner, B_HL, B_HR0, B_HR, and B_H2V planes:

   Each of these planes contains a single cell which serves as interconnect between blocks.

4.7.5 Master Cells

A set of master cells have been created in CMOS4S technology. Thus, the SC can generate a CLORA array in CMOS4S technology.

4.8 Summary

This chapter presents the implementation details of the folded PLA layout generator and the CLORA array layout generator. Having these two tools available, we can now evaluate our new FSM implementation method with the PLA implementation of FSMs. In the next

chapter, we will compare the new CLORA array FSM implementation with the PLA FSM implementation in terms of area.

CHAPTER 5

Results

The performance of the new FSM implementation method is evaluated by comparing the result of the PLA FSM implementation with that of the CLORA FSM implementation. FSMs that are used in the experiments including a stop watch controller, a test chip controller and several FSM examples provided by Microelectronics Center of North Carolina (MCNC). The information about these FSMs are presented in Table 5.1.

TABLE 5.1 Information of FSMs.

FSM Name	#. of States	#. of Inputs	#. of Outputs	#. of Edges
sw	16	2	4	32
test_chip	20	12	23	107
planet	48	7	19	115
dk16	27	2	3	108
dk512	15	1	3	30
lion	4	2	1	11
opus	10	5	6	22
ex4	14	6	9	21

The new FSM implementation method is evaluated for implementing both un-encoded and fully-encoded FSMs.

5.1 *clora*'s Results

This section presents the array sizes of the FSMs in Table 5.1 after being run through *clora*. In the following tables, gate rows refer to the rows that are occupied by a CLORA gate, and routing rows refer to rows that are used to route the inputs and the outputs into and out of the CLORA array. The pitch of a routing row is 64% of that of a gate row. Also presented in the tables is the execution time of Genie because the Genie process takes up most of the execution time in *clora*. The time given is the user time (i.e. time spent in the application program itself). Since the user time is not affected by the load on the system, it gives a better estimate on the actual execution time.

5.1.1 Un-encoded FSM Implementation

After running the above FSMs through *clora*, the array sizes of the un-encoded FSMs are given in Table 5.2 and Table 5.3.

TABLE 5.2 *clora*'s Results of the Un-encoded FSM Implementation With HR Plane.

FSM Name	Height (rows)		Width (cols)			Genie's Run Time (s)
	#. of Gate Rows	#. of Routing Rows	HL	HR	V	
sw	54	3	9	6	3	86.0
test_chip	164	11	24	6	12	15695.8
planet	184	7	23	10	12	11984.4
dk16	137	2	26	6	5	5623.6
dk512	45	1	11	4	5	147.3
lion	18	3	6	4	0	9.2
opus	40	6	11	4	5	93.9
ex4	51	6	9	5	5	96.3

TABLE 5.3 *clora*'s Results of the Un-encoded FSM Implementation Without HR Plane.

FSM Name	Height (rows)		Width (cols)		Genie's Run Time (s)
	#. of Gate Rows	#. of Routing Rows	HL	V	
sw	54	3	10	6	133.9
test_chip	180	11	21	13	4685.4
planet	201	8	19	26	7381.9
dk16	143	2	11	17	2030.3
dk512	51	1	7	6	102.2
lion	17	2	6	3	9.2
opus	47	7	10	6	85.2
ex4	57	6	11	7	102.5

Table 5.2 gives the array sizes of all the sample FSMs implemented in a conventional CLORA array described in Section 3.7. Table 5.3 gives the array sizes of all the sample FSMs implemented in a CLORA array without the HR plane. The reason for

experimenting with this variation is that we noticed that, in general, the device packing density of the HR plane is not very high. Thus, instead of having an HR plane to lay out low fan-in state gates and output gates, and a V plane to lay out high fan-in state and output gates, it may be more productive to have one single V plane to lay out all the state and output gates.

Comparing the results in Table 5.2 and Table 5.3, we see that, except for lion, the number of gate rows and the number of routing rows increased in the without-HR implementation. This is because in the without-HR implementation, the output gates are implemented by NORing either the 0-edges or the 1-edges depending on which one has the least number of edges. Since, in general, the number of 1-edges of an output is less than the number of 0-edges, more output gates are implemented by NORing the 1-edges. Thus, more inverters and more T-shaped output gates are required which leads to more gate rows and routing rows. In the case of lion, the only horizontal output gate in the with-HR implementation has changed into a vertical output gate in the without-HR implementation. Thus, its without-HR implementation has one less horizontal gate than its with-HR implementation.

On the other hand, we can also see that the widths of the CLORA arrays in Table 5.3 are narrower than those in Table 5.2. Apart from having one less horizontal plane, the reason for the narrower CLORA arrays is that the performance of genie is improved. This is because, with the without-HR implementation, we have only two separate planes instead of three. Thus, the result given by genie with the -s option is more effective.

While the with-HR implementation gives a shorter array, the without-HR implementation gives a narrower array. Thus, we cannot say which one gives a smaller FSM without considering the physical dimensions of the CLORA array (to be presented later).

5.1.2 Fully-encoded FSM Implementation

The following table presents the array sizes of the fully-encoded FSMs. These results are obtained by running *clora* with the -P option.

TABLE 5.4 *clora*'s Results of the Fully-encoded FSM Implementation.

FSM Name	Height (rows)		Width (cols)		Genie's Run Time (s)
	#. of Gate Rows	#. of Routing Rows	HL	V	
sw	36	2	10	7	102.8
test_chip	136	10	23	13	2634.9
planet	146	7	20	20	5992.3
dk16	90	2	14	6	1201.2
dk512	34	1	10	3	47.1
lion	16	4	7	2	3.8
opus	41	5	11	8	77.3
ex4	50	5	10	8	104.2

With fully-encoded FSMs, the number of state variables is decreased, and the logic expressions are optimized. Therefore, there are fewer gates in fully-encoded FSMs. Thus, as presented in Table 5.4, the heights of all the sample FSMs have been decreased significantly. On the other hand, the widths of all the sample FSMs are comparable to the ones in Table 5.3. This indicates that the amount of column sharing in fully-encoded FSMs is comparable to that of the un-encoded FSMs. This is because many of the inputs are used in almost every state in the FSM. This contradicts with our earlier belief that many of the inputs in FSMs are used only in a local set of states, and thus, in an un-encoded FSM, the net lengths are shorter, and can be folded more efficiently is not true.

5.2 Results of Physical Layouts

This section presents the area information of the various CLORA array layouts and PLA layouts of all the sample FSMs. In the following table, the area information is given in design micron square (DSM^2) in CMOS4S technology. Both the folded and unfolded PLA area are estimated as follows:

$$Area_{unfolded\ PLA} = W*H + N*6420 + 4.5*(3+2*N)*Max(L_D, W)$$

$$Area_{folded\ PLA} = W*H + N*6420 + 4.5*(3+2*N)*(W+ Max(L_D, H))$$

In the above equations, W corresponds to the width of the PLA; H corresponds to the height of the PLA; N corresponds to the number of state variables; and L_D corresponds to the length of the entire resettable D flip-flop block. The first component gives the area of the PLA. The second component gives the area occupied by the D flip-flops. Each resettable D flip-flop has an area of 6420 DSM^2. The third component gives the amount of routing area. The number of signals to be routed is computed in the second part of the third component. There are three control signals used by the flip-flops: Rb, Sb and CK. Then, there are the 2*N state variable signals that need to be routed. The width of the routing channel is estimated by multiplying the number of signals to be routed by the width of metal line pitch, 4.5 DSM. The length of the routing channel is estimated by the third part of the third component. For the unfolded PLA, all the I/O terminals are along the bottom side of the PLA. Assuming that the register block is placed along the bottom of the PLA, the length of the routing channel is estimated as the maximum between the length of the register block and the width of the PLA. For the folded PLA, the I/O terminals are on both the top and the bottom sides of the array. Assuming that the register block is placed along the left or right side of the PLA, the length of the routing channel is estimated as the sum of the width of the PLA and the maximum between the height of the PLA and the length of the register block.

TABLE 5.5 Actual Area of Various Implementations in DSM^2.

FSM Name	CLORA Array (un-encoded)		CLORA Array (fully-encoded)	PLA (fully-encoded)		CLORA (un-encoded) / PLA*	CLORA (fully-encoded) / PLA**
	With HR	Without HR		folded	unfolded		
sw	143027.2	137697.5	102958.2	147430.8	111494.5	1.23	0.92
test_chip	686825.3	652044.6	526937.0	533272.5	574354.1	1.22	0.99
planet	787220.5	892183.8	603541.8	677185.0	616748.0	1.28	0.98
dk16	506046.0	450912.0	238103.6	325177.0	269266.3	1.67	0.88
dk512	127522.7	116668.3	85064.1	128912.1	104724.4	1.11	0.81
lion	51955.7	47689.5	48058.9	62462.0	42424.8	1.12	1.13

TABLE 5.5 Actual Area of Various Implementations in DSM^2.

FSM Name	CLORA Array (un-encoded)		CLORA Array (fully-encoded)	PLA (fully-encoded)		CLORA (un-encoded) / PLA*	CLORA (fully-encoded) / PLA**
	With HR	Without HR		folded	unfolded		
opus	124190.5	130382.4	124407.6	160302.0	125769.5	0.99	0.99
ex4	144711.6	157638.0	141369.4	165809.5	148924.3	0.97	0.95

*: Ratio of the smallest PLA area and the smallest un-encoded CLORA array area.
**: Ratio of the smallest PLA area and the fully-encoded CLORA array area.

From Table 5.5, depending on the FSM, implementations with the HR plane performed both better and poorer than implementations without the HR plane. Thus, we cannot say which variation of CLORA array is better.

Also, shown in Table 5.5, the area of the un-encoded CLORA array implementation of the sample FSMs is on average about 1.2 times the area given by the fully-encoded PLA implementation. This is because the anticipated reduction in the width of the un-encoded CLORA FSM does not occur. Nonetheless, the CLORA arrays still give fairly reasonable results considering that they are implementing the un-encoded FSMs. In the case of ex4, the un-encoded CLORA implementation actually outdoes the fully-encoded PLA implementation. This shows that CLORA array is very efficient in implementing un-encoded FSMs.

On the other hand, the fully-encoded CLORA array FSM implementation is shown to be comparable to the fully-encoded PLA implementation.

As examples, several physical layouts of opus are shown in Figure 4.11-13.

FIGURE 4.11 Physical Layout of the With-HR Implementation of the Un-encoded Opus.

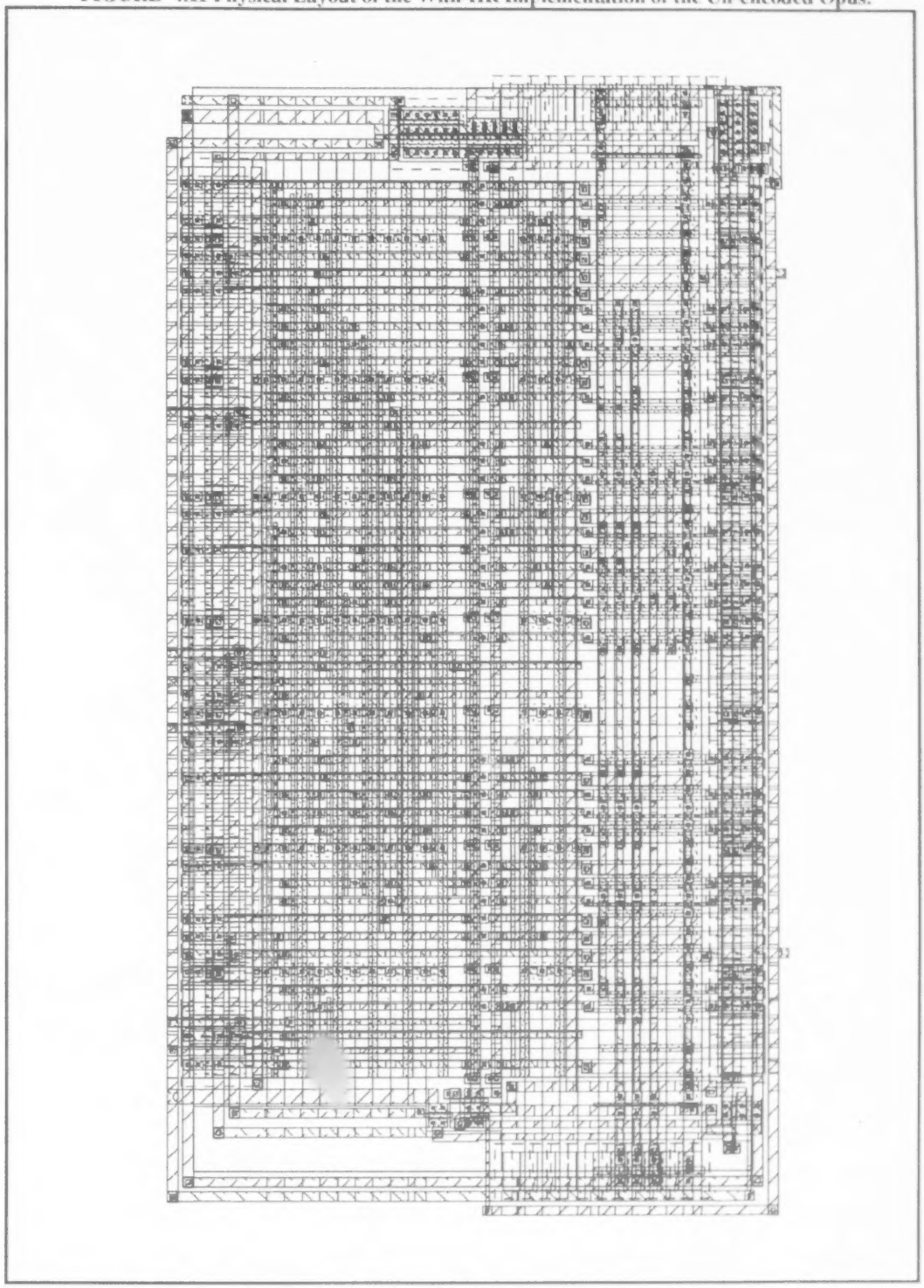


FIGURE 4.12 Physical Layout of the Without-HR Implementation of the Un-encoded Opus.

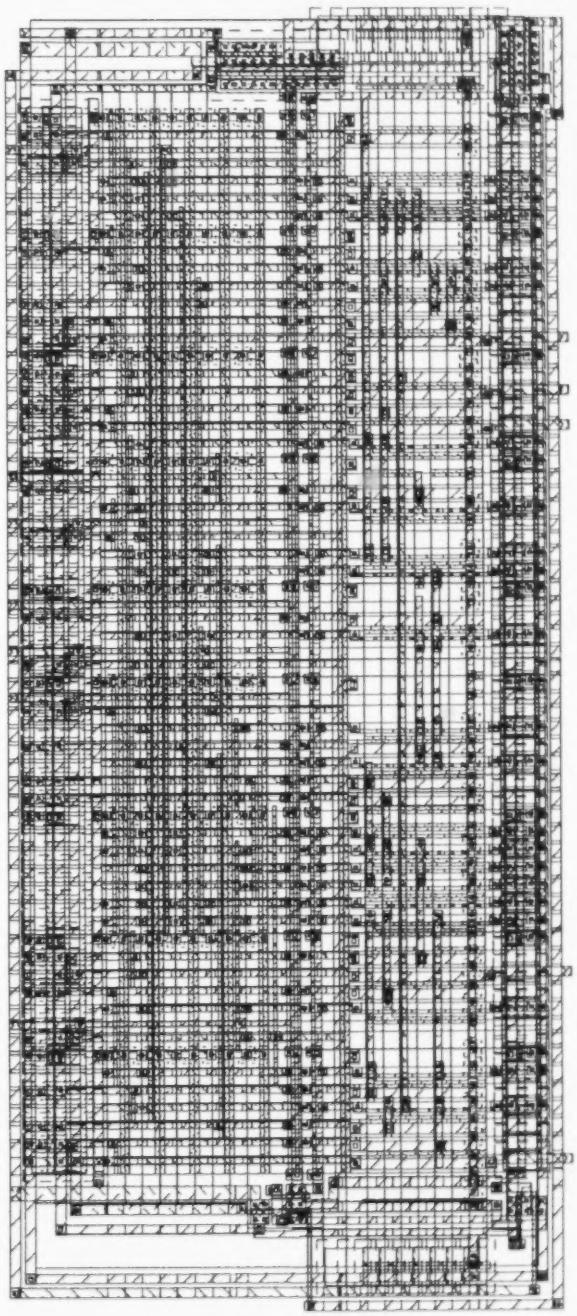
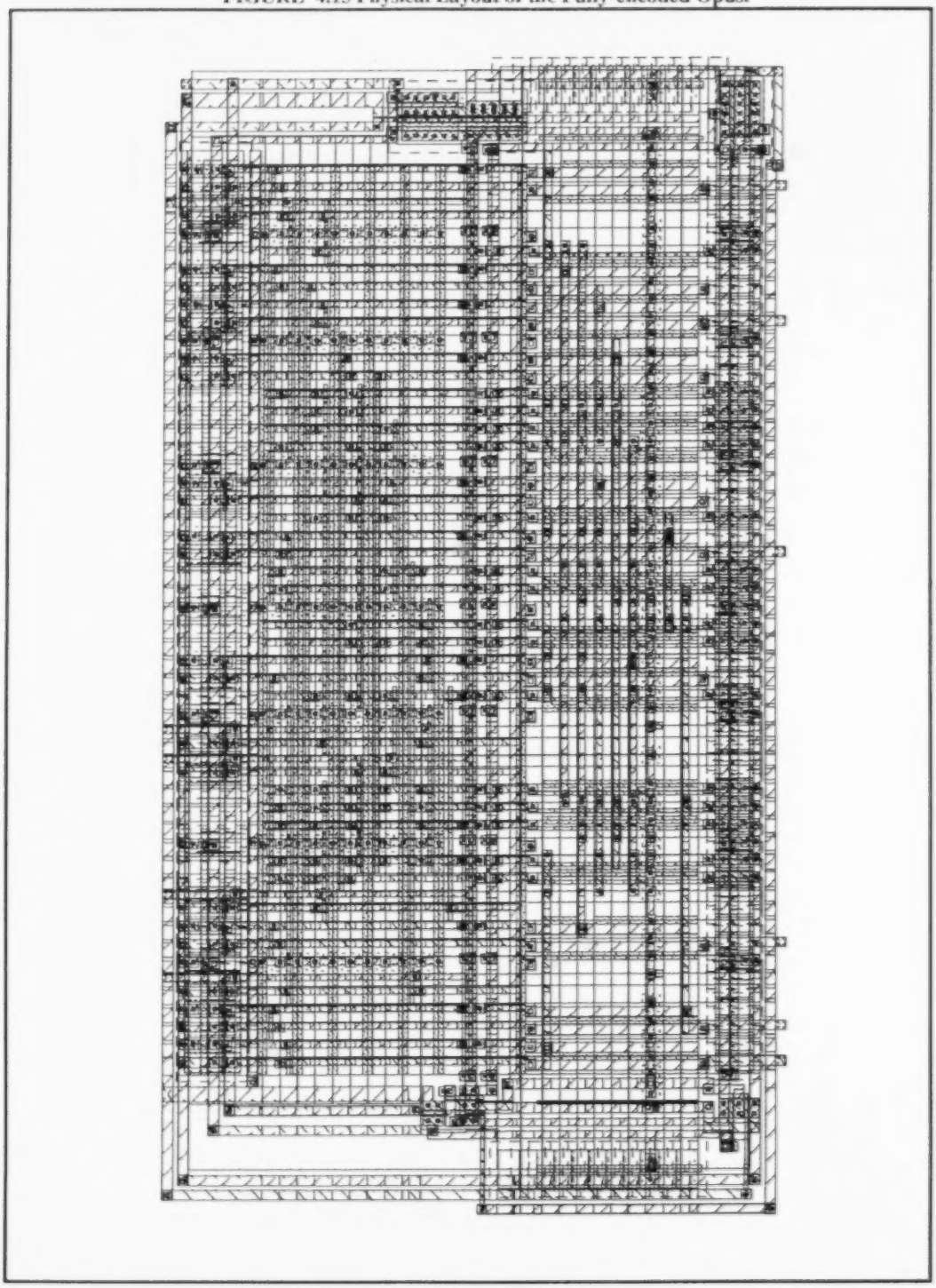


FIGURE 4.13 Physical Layout of the Fully-encoded Opus.



5.3 Speed Performance

The speed performance of the CLORA array was intended to be measured by testing the sw FSM chip that was sent to the Canadian Microelectronics Corporation for fabrication. However, due to a design error, the sw chip does not function properly. Thus, we measured the speed performance of the CLORA array by running an hspice simulation on the sw FSM layout. We found that the FSM can run at 80 MHz. This is fast enough for the current CMOS digital systems because the speed of the I/O pads is around 60 MHz.

5.4 Summary

This chapter presents the results obtained by implementing some of the sample FSMs using the new FSM implementation method. A comparison between the new CLORA array FSM implementation and the PLA FSM implementation in terms of area is done. It is found that the CLORA FSM on average give an area about 1.2 times that of a PLA implementation. Also, it is found by running hspice simulation that the CLORA FSM can run at 80 MHz.

CHAPTER 6

Conclusion

In this thesis, we considered the problem of implementing un-encoded FSMs. This problem is worth looking at because many of the FSM implementation methods only give reasonable sized FSMs when the FSMs are encoded. However, encoded FSMs have their limits. First, once the FSMs are encoded, the testability of the FSMs decreases. Second, FSMs that have concurrent operation sequences cannot be encoded. Thus, it is worthwhile to investigate the un-encoded FSMs implementation problem. Apart from considering un-encoded FSM implementation, we also considered the problem of merging the combinational logic and the memory elements together. In many of the FSM implementation methods, these two parts are treated as separate components and are connected together at the end of the implementation process. However, laying out FSMs this way, a significant amount of area is wasted in routing.

In order to come up with an efficient implementation method for un-encoded FSMs, the first thing to do is to eliminate the use of flip-flops or registers as the memory elements. This is because these cells are usually implemented as standard cells which take up a significant amount of area. To solve this problem, a new type of logic gate called CLOcked RAtioed logic (CLORA) is designed. The operation of a CLORA gate is straight forward. When the clock is high, the CLORA gate evaluates and behaves like a ratioed logic gate with its output voltage dependent on the aspect ratio of the N-type and the P-type transistors. When the clock is low, the CLORA gate behaves like a dynamic logic gate with its output value stored as charge at the output node. Using two sets of CLORA NOR gates controlled by two non-overlapping clock signals, an un-encoded FSM can be implemented. The first set of CLORA gates is responsible for evaluating the entrance conditions of all the states. The second set of CLORA gates is responsible for evaluating all the state variables and outputs. The layout structure of the CLORA FSM is chosen to be in a regular array structure because it is easy to automate.

The generation of a CLORA FSM involves executing two programs. The first one is a C++ program called *clora*. This program is responsible for transforming the state transition table of an FSM into a symbolic representation of the physical layout of the FSM. The second program is an IL program called CLORA(). This program is run under the Cadence design environment and is responsible for generating the physical layout based on the symbolic representation generated by *clora*.

In order to evaluate the performance of this new FSM implementation method, several sample FSMs provided by MCNC were laid out using this new method. The area of these CLORA FSMs was compared with that of the PLA implementations of these FSMs. We found that on average the area of an un-encoded CLORA FSM is about 1.2 times that of a fully-encoded PLA implementation. This indicates that the CLORA FSM implementation is fairly efficient in implementing un-encoded FSMs. Thus, this new method may offer an alternative in implementing concurrent FSMs. Moreover, by running hspice simulation on one of the sample FSMs, we found that the CLORA FSM can run at 80 MHz which is fast enough for the current CMOS digital circuits.

There are two main reasons that the CLORA FSM gives larger area than the PLA implementation. The first reason is that since one-hot state assignment is used, there are more product terms to implement. Thus, there are more gates in the un-encoded CLORA FSM, and this results in the CLORA FSM being much taller than the PLA implementation. The second reason is that the expected reduction in the width of the CLORA array does not occur. This is because the inputs of the sample FSMs are used in almost every state. Thus, our earlier belief that many of the FSM inputs are used only in a local area is not true. Nonetheless, the CLORA FSM implementation method still gives fairly reasonable results in implementing un-encoded FSMs. Future work on CLORA FSM includes finding efficient ways to reduce the height and the width of the CLORA array. The height of the array can be reduced by making use of the multi-level capability of the CLORA array and optimizing the FSM logic expressions so that the number of

gates is reduced. The width of the array can be reduced by finding a better way to arrange the order of the gates such that column folding can be more effective.

References

1. R. Z. Makki, J. Muha, S. Boughazale, and T. Kaylani, "SSC - A Tool for the Synthesis of Testable Sequential Machines", *Digest of papers - Thirty-Fifth IEEE Computer Society International Conference - COMPCON 89*, pp. 455-461, 1990.

2. D. J. Comer, *Digital Logic And State Machine Design*. Saunders College Publishing, 1990.

3. S. Devadas, H. T. Ma, A. R. Newton, and A. Sangiovanni-Vincentelli, "MUSTANG: State Assignment of Finite State Machines Targeting Multilevel Logic Implementations", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 7, no. 12, pp. 1290-1300, Dec. 1988.

4. R. K. Brayton, R. Rudell, and A. Sangiovanni-Vincentelli, "MIS: A Multiple-Level Logic Optimization System", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. CAD-6, no. 6, pp. 1062-1081, Nov. 1987.

5. G. D. Micheli, R. K. Brayton, and A. Sangiovanni-Vincentelli, "Optimal State Assignment of Finite State Machines," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. CAD-4, pp. 269-285, July. 1985.

6. T. Villa, and A. Sangiovanni-Vincentelli, "NOVA: State Assignment of Finite State Machines for Optimal Two-Level Logic Implementation", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 9, no. 9, pp. 905-924, Sept. 1990.

7. R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*. New York: Kluwer Academic, 1984.

8. W. I. Fletcher, *An Engineering Approach to Digital Design*. Englewood Cliffs, N. J.: Prentice-Hall, 1980.

9. J. E. Lecky, O. J. Murphy, and R. G. Absher, "Graph Theoretic Algorithms for the PLA Folding Problem", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 8, no. 9, pp. 1014-1021, Sept. 1989.

10. A. D. Lopez, and H. S. Law, "A Dense Gate Matrix Layout Method for MOS VLSI", *IEEE Journal of Solid-State Circuits*, vol. SC-15, no. 4, pp. 736-740, Aug. 1980.

11. S. Huang, and O. Wing, " Gate Matix Partitioning", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 8, no. 7, pp. 756-767, July 1989.

12. S. M. Sait, and M. A. Al-Rashed, "Efficient Algorithm for Weinberger Array Folding", *International Journal of Electronics*, vol. 69, no. 4, pp. 509-518, Oct. 1990.

13. N. Weste, and K. Eshraghian, *Principles of CMOS VLSI Design*. Addison-Wesley, 1985.

14. R. Amann, and U. G. Baitinger, "Optimal State Chains and State Codes in Finite State Machines", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 8, no. 2, pp. 153-170, Feb. 1989.

15. *Cadence Reference Manuals*, Cadence Design Systems, Inc., 1989.

16. *Module Generation in the Cadence Design Environment-User's Manual*, University of British Columbia.

17. *Octtools-5.1 Manuals*, Electronics Research Laboratory, University of California, Berkeley, Oct. 1991.

18. S. Devadas, and A. R. Newton, "Topological Optimization of Multiple-Level Array Logic", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. CAD-6, no. 6, pp. 915-941, Nov. 1987.

19. S. Baase, *Computer Algorithms (Introduction to Design and Analysis)*. Addison-Wesley, 1988.